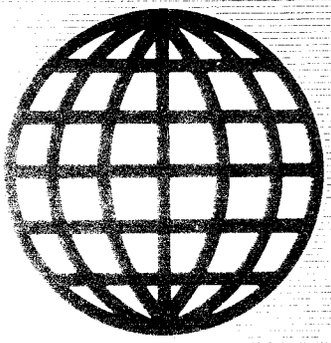


The COMPUTER JOURNAL®

Programming - User Support
Applications

Issue Number 43 | March / April 1990 | $3.00

Standardize Your Floppy Disk Drives

A New History Shell for Z-System

Heath's HDOS, Then and Now

The Z-System Corner

Graphics Programming With C

Lazy Evaluation

S-100 There's Still Life in the Old Bus

Advanced CP/M

The NS32000

The Computer Corner

The Computer Journal

Editor/Publisher
Art Carlson

Art Director
Donna Carlson

Circulation
Donna Carlson

Contributing Editors
Bill Kibler
Bridger Mitchell
Clem Pepper
Richard Rodman
Jay Sage
Dave Wenstein

The Computer Journal is published six times a year by Technology Resources, 190 Sullivan Crossroad, Coulmbia Falls, MT 59912
(406) 257-9119

Entire contents copyright © 1989 by Technology Resources.

Subscription rates—$16 one year (6 issues), or $28 two years (12 issues) in the U.S., $22 one year in Canada and Mexico, and $24 (surface) for one year in other countries. All funds must be in U.S. dollars on a U.S. bank.

Send subscription, renewals, address changes, or advertising inquires to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, MT 59912, phone (406) 257-9119.

The COMPUTER JOURNAL

Issue Number 43 March / April 1990

Editorial 2

Standardize Your Floppy Disk Drives 4
Using Dysan's Diagnostic to check your floppy drives. By Eugene L. Langberg.

LSH 11
A new history shell for Z-System. By Rob Friefeld.

Letwin's Prior Progeny 14
Heath's HDOS, then and now.
By Kirk L. Thompson.

The Z-System Corner 17
Software Update Service, and customizing NZCOM. By Jay Sage.

Graphics Programming With C 19
Writing graphics routines for the IBM PC, and the Turbo C graphics library. By Clem Pepper.

Lazy Evaluation 26
How to save time by ending the evaluation of logical expressions as soon as the result is known. By Marla Bartel.

S-100 28
There's still life in the old bus. Installing a new disk controller and video board. By Michael Broschat.

Advanced CP/M 31
Passing parameters when space is at a premium and complex error recovery must be managed.
By Bridger Mitchell.

Real Computing 33
The NS 32000. By Richard Rodman.

The Computer Corner 40
By Bill Kibler.

Editor's Page

The Challenge of the Future

The computer industry is continuing its fast paced change with more powerful software and hardware announced daily. No one can accurately predict what will be available next year, and there is a lot of discussion about whether companies should buy now or if they should wait till the more powerful systems are available. While a lot of attention is given to the future of computer systems, very little is said about the future of the people.

It is time to think about where people will fit in with the future computer systems. Will there be enough jobs for programmers and hardware designers? And if so, will they be in what we normally consider the general computer industry, or will they be in industrial applications? If someone entering college asked for your advice on selecting a curriculum to prepare them for the job market five years from now, would you recommend courses on hardware, software, or business applications of computers?

I feel that there will be a decreasing demand for programmers and hardware designers in the high volume general computer field starting by 1995. I base this on the following assumptions.

The new systems are too large and complicated for an individual to be effective in programming or hardware design. It takes a large company to provide sufficient resources.

The business users aren't interested in computers, they are only interested in what it can do for their business. The buyers demand well polished software with high quality manuals plus on-site support and training classes. They also want multiple sources so that they can use competitive bidding to force the prices down. They are only comfortable in working with other large companies who "know how to play the game."

Most of the products will come from large organizations, and any large organization tries to manage for the maximum short term profit. This means that they will adopt CAE (Computer Aided Engineering), Application Generators, and any other tools in order to eliminate the expense of hiring people. It does not matter that the resulting designs are less efficient than what could be done by knowledgeable engineers, after all, hardware is cheap, and they'll just use larger faster systems.

Anything which can not be automated will be moved out of the United States. Most of the hardware manufacturing has already been either automated or moved. Software programming is currently being sent off-shore (a lot of it to India), and more will be sent in the near future. There is some hardware design being done off-shore and this trend will accelerate.

We will be left with upper management groups who will decide what hardware or software is needed, and then send it out for either automatic generation and design, or overseas for design and production--of course the people overseas also recognize the potentials of automation, so a lot will be sent overseas for automatic generation and design.

As other countries become more proficient they will also become effective in over all market design strategies. Then they will ship us their own finished products which they designed, and we'll be left out of the picture except to send them our dollars. You say that it couldn't happen? Talk to someone in the auto or steel industry. Or figure out how many TVs or VCRs are designed and made here. When was the last quality camera designed or made in the US?

The computer industry is now at the point of change, and I believe that we are entering the final phases of the individual computer related entrepreneur. Most hardware and software design will be automated or sent overseas, so there will be very few customers for the type of products which could be profitably marketed by an individual. There will be some niche exceptions such as database programming, but even these are being automated (Paradox 3.0 and FoxPRO) so that the user can develop their own applications without actually programming. There will be many openings for people who understand business computer applications, but few openings for programmers.

I would recommend that the student study business, Lotus 1-2-3, and Novell NetWare, rather than computer design or programming in C or dBASE.

Where does this leave those of us who enjoy programming or hardware development? There are two viable possibilities. 1) Earn your income in some other field, and use computers as a hobby where you can do what you enjoy. 2) Train yourself for the few computer related employment opportunities which still exist, especially if these are in an area which you enjoy.

One area which I find fascinating and which still provides significant opportunity is the design of embedded process controller applications. Each application is so specific and so demanding that it still takes a sharp human brain to devise a solution. This is the one field which I would recommend to someone who insisted that they wanted to earn a living working on computers (as contrasted with office functions which work **with** computers).

I am expanding the electronics bench, and will switch most of my work towards hardware design and programming of controllers, some of which will be merely for fun. One of the fun projects, which still

provides a good opportunity for learning, is a model train layout with smart controllers on the train(s) and infrared communications between the train(s) and the base station. I will have no loyalty as to what type of computer I work with--they are only an appliance. If I need PageMaker or Schema II I'll use the '286 because that's what they run on. I'll use what ever a cross assembler runs on, but it might not be the same system I use for writing the assembler source code. It will be very rare that my target system will be a traditional computer system, although I may use an S-100 system as a master with dedicated processors on the boards.

While I feel that there is only a very limited future in working on (remember working on is very different than working with) traditional systems, you are entitled to disagree. I welcome your comments, rebuttals, and poison pen letters.

The Computer Publishing Market

The computer book publishing industry is in trouble (see editorial in #40). They published too many titles for the wrong market, too many poorly written books, and tried to distribute technical titles through the mass markets. Now, they are retrenching and making drastic reductions in their list.

We have received a notice from Howard W. Sams which states in part, "...shares your concerns about possible overpublishing in the computer books segment of the industry. We strive to meet retailers' needs by maintaining a short, high-performance list." The enclosed Out of Print Notice lists 192 titles which are being dropped! The good news is that Lancaster's TTl and CMOS Cookbooks are still in print.

We are headed into a dearth of technical titles, because these titles are not suitable for distribution through the high volume mass market channels. It is very important that we encourage and support the companies which still serve our needs. We will have to utilize mail order resources to obtain our books because they will not be stocked at Dalton's or Waldenbooks. The time is ripe for someone to start a well publicized mail order outlet for technical books from various publishers. It should be located in a rural low-overhead area, and be strictly mailorder to avoid the expenses associated with a retail storefront. They could get started by buying a few select remainders from Sams' out of print list (remainders are purchased at a small fraction of the original price).

I know of three companies currently publishing useful technical titles. One is Tab Books (Blue Ridge Summit, PA 17294-0850, (717) 794-2191). I have just received their book "The Programmable Logic Device Handbook" by Burton, which covers a vital new topic. I also want to see "Programmable Controllers: Hardware, Software, and Applications" by Batten, and "Brushless DC Motors: Electronic Commutation and Controls" by Sokira and Jaffe. They have dozens of good titles, don't fail to write for their catalog.

A second company is M&T Books (501 Galveston Drive, Redwood City, CA 94063 1-800-533-4372 (in CA 1-800-356-2002)). M&T also publishes Dr. Dobbs Journal. One of their titles is "Graphics Programming in C" by Stevens (see Pepper's article in this issue). Another recent title is "Building Local Area Networks with Novel's NetWare" by Corrigan and Guy. Most of their books are available with disk. Write for their catalog.

The third company is Addison-Wesley (Route 128, Reading MA 01867, (617) 944-3700). They publish four books on

(Continued on page 38)

SAGE MICROSYSTEMS EAST

Selling & Supporting the Best in 8-Bit Software

- New Automatic, Dynamic, Universal Z-Systems
  - Z3PLUS: Z-System for CP/M-Plus computers ($69.95)
  - NZ-COM: Z-System for CP/M-2.2 computers ($69.95)
  - ZCPR34 Source Code: if you need to customize ($49.95)
- Plu*Perfect Systems
  - Backgrounder II: switch between two or three running tasks under CP/M-2.2 ($75)
  - ZDOS: state-of-the-art DOS with date stamping and much more ($75, $60 for ZRDOS owners)
  - DosDisk: Use DOS-format disks in CP/M machines, supports subdirectories, maintains date stamps ($30 - $45 depending on version)
- BDS C — Special Z-System Version ($90)
- SLR Systems (The Ultimate Assembly Language Tools)
  - Assembler Mnemonics: Zilog (Z80ASM, Z80ASM+), Hitachi (SLR180, SLR180+), Intel (SLRMAC, SLRMAC+)
  - Linkers: SLRNK, SLRNK+
  - TPA-Based: $49.95; Virtual-Memory: $195.00
- NightOwl Software MEX-Plus ($60)

Same-day shipping of most products with modem download and support available. Order by phone, mail, or modem. Shipping and handling $4 per order (USA). Check, VISA, or MasterCard. Specify exact disk format.

Sage Microsystems East
1435 Centre St., Newton Centre, MA 02159-2469
Voice: 617-965-3552 (9:00am - 11:30pm)
Modem: 617-965-7259 (password = DDT)(MABOS on PC-Pursuit)

Standardize Your Floppy Disk Drives

by Eugene L. Langberg

Last fall I put together an IBM PC/XT Clone from case, power supply, boards and floppy disk drives purchased at a Computer Show and Flea Market and through mail order. The machine went together easily and worked on the first try with no apparent problems.

The machine proved able to run all sorts of IBM software without exception both public domain and copyright material. The extensive testing with IBM software convinced me that the machine is an excellent clone, truly compatible.

The time had come to get down to some serious work. I bought a copy of Flight Simulator and tried it. The A drive made its usual noise, its LED went on and off a few times then all was quiet. The monitor's screen was blank, there was no response to keyboard input, the machine had blocked-up.

What could be wrong? It just had to be the software. A quick call for help to a friend who has an IBM PC/XT allowed me to try my Flight Simulator program on the real machine. It loaded and ran. There was nothing wrong with the software. The problem was in my clone, but where?

After going over connections etc. with great care, and again trying lots of IBM software, including IBM's diagnostics, none of which failed, I became convinced that the problem was in my floppy disk drives. They appeared to be unable to read the data from the Flight Simulator diskette. I substituted a disk drive taken out of a CP/M machine for the clone's drive A. My assumption proved to be correct. The substitute drive was able to read all of the software the troublesome drives could read and Flight Simulator as well. My clone was compatible, my floppy disk drives were not.

What could be wrong with a pair of apparently new drives that could read lots of different software diskettes, without error, but were unable to read the data from one particular diskette? My conclusion, the drives required realignment.

Floppy disk drive designs are, for the most part, highly reliable. Computer operators use their drives day in and day out for years without special care or problems. Software houses make thousands of floppy disk copies of their software, sell and send these copies to users all over the world without any doubt in mind that the user will be able to read and use the material on the disks they receive. This is remarkable and attests to the reliability of both floppy disks and floppy disk drives. However, in order for the disks and drives to achieve this high degree of compatibility for information interchange, they both must adhere closely to universally accepted standards of recording and adjustment.

There are two simple adjustments that you can make to your floppy disk drives using your computer and simple tools if you have the proper software. These adjustments are the disk speed and track alignment. However, unless you are in the business of repairing floppy disk drives and therefor have the tools, specifications, access to replacement parts and the know-how for making mechanical repairs, it is best that you not attempt to make major repairs to your drives.

There are a number of programs in the public domain which enable you to check the speed of your disks by simply running the programs and making note of the results shown on the video screen. Many floppy disk drives have a small adjustable resistor on their printed circuit board which is used to set the speed. One adjusts this resistor with a small screw-driver and repeats the test until you have set the drive to a speed as close to 300 RPM as you can. Some drives even have a stroboscopic pattern on their pulleys to facilitate this adjustment without using special software.

Checking track alignment is another matter. Doing so requires that one have available a diskette made specifically for this purpose. There are two kinds of such diskettes available. One is analog, the other is digital. To use the analog diskette one must have a dual trace oscilloscope with specifications such, that it in itself becomes quite expensive. Not at all cost effective if one wishes to check a few drives once in a while. On the other hand the digital diskette, costing no more than some inexpensive software, is a worth while investment for occasional checking of drives.

Dysan makes a series of Digital Diagnostic Diskettes which can test drives in a computer using no special test equipment. The diskettes are available in single-sided single-density, single-sided double-density, double-sided single-density and double-sided double-density. I purchased the double-sided double-density diskette, model number 508-400 for use with my IBM PC/XT clone. The list price for this diskette is $40.00.

By use of the Dysan Digital Diagnostic diskette and appropriate software, the following are some of the tests that can be made on a floppy disk drive without removing it from the computer.

Head Radial Alignment
Head Positioner Linearity
Head Positioner Hysteresis
Diskette-Clamping Eccentricity
Head-to-Media Compliance
Index/Sector Photo-Detector timing
Head Positioner Skew
Diskette Rotational Speed
Head Azimuth Alignment
Head Load Actuator Timing

All of the above are tests of the mechanical adjustments of a drive. Drives purchased from their manufacturer should be able to pass these tests without question and hold the adjustments despite long and hard use. However if, as were mine, your drives were purchased from someone who deals in drives with an unknown history, the above tests can give you information about their condition and adjustment. However, the fact remains, if you are not in the business of drive repair the only practical adjustments you can make to your drives is the rotational speed and the track alignment or Head Radial Alignment as Dysan refers to it.

The model 508-400 Digital Diagnostic Diskette has data recorded on tracks 0, 5, 16, 19, 30 and 39 in progressive offset. Other tracks have data recorded on them differently. For Head Radial Alignment testing the progressive offset tracks are used. The tracks are written with all the track and sector ID fields on track. However, the data fields are written radially displaced from the track centerline. The data field of sector one is displaced 6 thousandths of an inch toward the spindle, that of sector two, 6 thousandths of an inch away from the spindle. The data fields of each successive pairs of sectors are displaced an additional thou-

sandth of an inch toward and away from the spindle.

The read/write heads of a 5¼ inch disk drive are 12 thousandths of an inch wide. A drive with perfect Head Radial Alignment and excellent head sensitivity should be able to read data from alternate sectors out to the last sectors where data is recorded within the 12 thousandths of an inch head path. This would be for all sectors out to 11 and 12. The program we must have to test Head Radial Alignment must permit us to read all readable sectors on the selected tracks that have the progressive offset.

However, there is a problem if the program is to run under PC-DOS. The Dysan Diagnostic Diskettes are formatted with 16 sectors per track, and 256 bytes per sector for the double-density models. This is not the PC-DOS format used by the IBM PC or compatible machines which use diskettes formatted 8 or 9 sectors per track and 512 bytes per sector, therefor, special software must be written to read and use the diagnostic diskettes.

There are two ways to solve this problem. One is to write our own disk-base, a table maintained by the operating system in the BIOS of the PC, which contains the disk parameters including the number of sectors per track and the number of bytes per sector, and temporarily replace the standard disk-base in memory with one whose disk parameters include 16 sectors per track and 256 bytes per sector. Doing this will allow BIOS INT 13H and its read function, AH = 2, to read the Dysan Diagnostic Diskette. The second way to handle this problem is to by-pass the BIOS and directly address the Floppy Disk Controller chip.

The first program I wrote to do this job involved the replacement of the disk-base table and use of BIOS INT 13H. However, I did not like the way the program operated. The program was designed to read all of the sectors from the selected track. However, it would not read through a CRC error. This made for a rather coarse test. A reworked program that would read a sector, report an error if one had occurred and would then read the next sector, was tried and found cumbersome to use.

The program was completely rewritten to read the data from the diskette by addressing the Floppy Disk Controller chip directly. It is this program that has been included with this article.

The program makes use of a function available in the controller chip that is not available through the PC BIOS. This function reads all of the data stored in all of the sectors of the selected track ignoring any CRC errors that may be encountered. In our application the drive will attempt to read data from all the sectors of the progressive offset tracks. If the data is so far offset that it can not be read or is misread, the process will not halt despite CRC errors but will continue on to the end of the track. When the data that has been read is examined one can see where the drive begins to drop bits and where the drive can no longer read anything from the offset sector.

The program is not complicated nor tricky but in order to see how it works does require some understanding of how one addresses the very intelligent Floppy Disk Controller (FDC) and Direct Memory Access (DMA) chips and sends commands to them.

The program comes up with a message that explains how to exit the program, how to terminate the display of data and restart the program and how to toggle the display so that it stops and starts to allow the examination of the data in particular sectors before they scroll

```
;TITLE Floppy Disk Drive Head Radial Alignment Program

;Author:  Eugene L. Langberg
;Date: June 21, 1987

;Requirements for use:
;  An IBM PC/XT with one or more Floppy Disk Drives
;  A Dysan Digital Diagnostic Diskette, Model No. 508-400

;MACRO AREA

PUSHR  MACRO        ;THESE MACROS REPRESENT THE SHOTGUN
   PUSH   AX     ;WAY OF SAVING REGISTERS SO AS NOT
   PUSH   BX     ;TO WORRY IF YOU HAVE SAVED THE CORRECT
   PUSH   CX     ;ONES OR NOT WHEN YOU MOVE ABOUT IN
   PUSH   DX     ;THE PROGRAM BY CALLS TO ROUTINES THAT
   PUSH   BP     ;DO LIMITED THINGS SUCH AS OUTPUT A
   PUSH   DI     ;CHARACTER.
   PUSH   SI     ;PUSHR SAVES A GROUP OF REGISTERS
   ENDM

POPR   MACRO        ;POPR RESTORES THE SAME REGISTERS
   POP SI
   POP DI
   POP BP
   POP DX
   POP CX
   POP BX
   POP AX
   ENDM

DATA SEGMENT

;MESSAGE AREA

SIGNON DB   13,10,10
   DB   '(CTRL C) TERMINATES THE PROGRAM, (CTRL A) TERMINATES THE',13,10
   DB   '  DISPLAY AND REPEATS THE PROGRAM AND (ESC) TOGGLES THE  ',13,10
   DB   '    DISPLAY STOPPING AND THEN STARTING IT AS YOU WISH ',
   DB   13,10,10,'$'
MSG_DRV DB   '      ENTER THE SELECTED DRIVE (0 FOR A, 1 FOR B)             $'
MSG_TRK DB   '             ENTER TRACK NUMBER (0 - 39)                      $'
MSG_HD DB   '                 ENTER HEAD (0 -1)                           $'
ERR_MSG DB   13,10
   DB   ' DIGITS ONLY IN THE RANGE SPECIFIED, TRY AGAIN   ',13,10,10,'$'
TRK_MSG DB   'TRACK $'
HD_MSG DB '  HEAD $'
SEC_MSG DB   '  SECTOR $'

;DECLARED VAIABLE AREA

DRIVE     DB ?
TRACK     DB ?
HEAD      DB ?
SECTOR    DB ?
DRIVE_HEAD   DB ?
MOTOR_ON  DB ?
MOTOR_OFF DB ?
DATA_BUFF DB 4096   DUP(9FH) ;AN ODD FILL CHARACTER
DECIMAL_BUFF DB 3   DUP(0)     ;STORE FOR DECIMAL NUMBERS

SECTOR_SIZE  DB 001H          ;256 BYTE SECTORS
LAST_SECTOR  DB 010H          ;16 SECTORS PER TRACK
GAP3_LENGTH  DB 021H          ;GAP-3 LENGTH FOR DYSAN READ
DATA_LENGTH  DB 0FFH          ;USE 0FFH SINCE SECTOR SIZE IS SPECIFIED

DATA   ENDS

STACK SEGMENT STACK
       DB 80H DUP (?)         ;A STACK OF 128 BYTES
STACK  ENDS

CODE SEGMENT
   ASSUME CS:CODE,DS:DATA

MAIN   PROC   FAR
   MOV AX,DATA    ;POINT DS TO THE
   MOV DS,AX      ;DATA SEGMENT

START: MOV AH,0H      ;RESET THE DISK SYSTEM
   INT 13H    ;WITH BIOS INTERRUPT CALL
   CALL   CLRSCR     ;CLEAR THE SCREEN
   MOV DX,OFFSET SIGNON ;DISPLAY THE SIGN ON MESSAGE
   CALL   MSGOUT
   CALL   SETUP      ;INPUT SELECTED DRIVE, HEAD AND TRACK
   CALL   CLRSCR     ;CLEAR THE SCREEN AGAIN
   CALL   READ_TRACK          ;READ THE DATA ON THE SELECTED TRACK
   CALL   SHOW       ;DISPLAY WHAT THE DRIVE WAS ABLE TO READ
   CALL   DELAY      ;HOLD THE DISPLAY FOR A WHILE
   JMP START      ;BEFORE RESTARTING THE PROGRAM

MAIN   ENDP

;THE FLOPPY DISK CONTROLLER (FDC) OPERATES THROUGH 3 I/O PORTS
;  03F2H  THE DIGITAL OUTPUT REGISTER
;  03F4H  THE STATUS REGISTER
;  03F5H  THE DATA REGISTER
;
;THE READ_TRACK PROCEDURE DOES THE FOLLOWING:
;TURNS ON THE MOTOR OF THE SELECTED DRIVE
;WAITS FOR IT TO COME UP TO SPEED
;RECALIBRATES THE SELECTED DRIVE
;DOES A SEEK TO THE SELECTED TRACK
;WAITS FOR AN INTERRUPT THAT INDICATES THE SEEK IS DONE
;INITIALIZES THE DMA TO MOVE DATA TO MEMORY
;SENDS READ INSTRUCTIONS TO THE FDC
```

away. The program then asks for input from the keyboard. The drive to read from, the head or side of the disk to read and the track to read. After this information is entered there is a short wait while the motor of the selected drive comes on, the drive is recalibrated (its head goes to track zero and the controller is reset), the drive does a seek (the heads move to the selected track), the drive reads the data from all the sectors on the selected side (head) into a buffer in memory via DMA, turns the motor off and displays the contents of the buffer on the video monitor in clearly marked sectors. There is a delay after the sixteenth sector has been shown and the program restarts.

The Floppy Disk Controller chip used in the IBM-PC is a NEC 765. The 8272A FDC chip is the same. The FDC is directly addressed through three I/O ports. They are:

```
03F2H     The Digital Output Register
03F4H     The Status Register
03F5H     The Data Register
```

The eight bits of the Digital Output Register have the following meaning:

```
bits 1-0  select the drive, 00 = A, 01 = B, 10 = C, 11 = D
bit 2     when 0 the drive resets
bit 3     when 1 it enables the FDC interrupt and DMA access
bits 7-4  turn on motor, 0001= A, 0010= B, 0100= C, 1000= D
```

Turning a specific drive on or off is simply a matter of sending the proper byte of information to the Digital Output Register. A byte whose bits are 00011100B = 28., sent to the Digital Output Register will turn on drive A, enable its interrupt and enable DMA access to it, for drive B the byte is 00101101 or 45. To turn drive A off and reset it the byte 00000000 or 0. is sent to the Digital Output Register, for drive B the byte is 00000001 or 1.

The eight bits of the Status Register have the following meaning:

```
bits 3-0  tell which drive is in seek mode,  0001 = A,
          0010 = B, 0100 = C and 1000 = D.
bit 4     1 = a read or write is in progress
bit 5     1 = FDC is not in DMA mode
bit 6     1 = FDC Data Register is ready to send data
          0 = FDC Data Register is ready to receive data
bit 7     1 = FDC is ready to send or receive data
```

Data being sent to or being received from the drive is placed in the Data Register a byte at a time. When the drive is in DMA mode, data is transferred through this register into or from memory directly without CPU involvement. When the drive is not in DMA mode the CPU must transfer data to or from the drive through this register one byte at a time as well. The non DMA mode is often used for sending control commands to peripheral devices. In the accompanying program the DMA mode is used to transfer data that has been read from the diskette into memory. It is then transferred from memory to the screen of the video display.

The program's short procedure OUT_FDC demonstrates how data is sent to the FDC in the non DMA mode. We need only read the Status Register repeatedly until bit seven is turned on to know when a byte of data should be sent to the Data Register. The same procedure is used to read data from the FDC, when not using DMA mode, but the instruction OUT is replaced with the instruction IN.

The control commands that are sent to the Digital Output Register do not require reading the Status Register to see if the ready bit is on. However, these commands are limited in number and might be thought of as a first level of command. One certainly cannot read or write to a drive whose motor has not been turned on. Once a drive has been turned on, the intelligence of the FDC chip will permit the execution of no fewer than fifteen commands. These commands are sent to the FDC through the data register.

The procedure used to send commands to the FDC is simple. An operation code is the first of a series of bytes sent to the FDC. This alerts the FDC intelligence that a command is coming in.

```
;WAITS FOR AN INTERRUPT THAT INDICATES THE DATA READ IS IN MEMORY
;TURNS OFF THE MOTOR
;
;* MUCH OF THIS ROUTINE, WITH MODIFICATIONS, HAS BEEN TAKEN FROM
;  ROBERT JOURDAIN'S BOOK ''PROGRAMMER'S PROBLEM SOLVER FOR
;  THE IBM PC, XT & AT'', SECTION 5.4.1

READ_TRACK    PROC    NEAR

;TURN ON THE SELECTED MOTOR

    STI         ;BE SURE THE INTERRUPTS ARE ENABLED
    MOV DX,03F2H  ;FDC DIGITAL OUTPUT REGISTER PORT ADDRESS
    MOV AL,MOTOR_ON  ;SET PROPER BITS FOR THE SELECTED MOTOR
    OUT DX,AL      ;SEND THE COMMAND TO THE FDC

;WAIT FOR MOTOR TO COME UP TO SPEED

        MOV     CX,3500       ;COUNT FOR AN EMPTY LOOP (PC OR XT)
MOTOR_DELAY: LOOP    MOTOR_DELAY ;IDLE FOR 1/2 SECOND

;RECALIBRATE THE SELECTED DRIVE

RECAL: MOV AH,07H     ;OP-CODE FOR DRIVE RECALIBRATION
    CALL   OUT_FDC    ;SEND THE OP-CODE TO THE FDC
    MOV AH,DRIVE   ;DRIVE TO RECALIBRATE
    CALL   OUT_FDC    ;SEND IT TO THE FDC
    CALL   WAIT_INTERRUPT       ;WAIT FOR THE OPERATION TO COMPLETE

;DO A SEEK TO THE SELECTED TRACK

SEEK:  MOV AH,0FH     ;OP-CODE NUMBER FOR SEEK
    CALL   OUT_FDC    ;SEND THE OP-CODE TO THE FDC
    MOV AH,DRIVE_HEAD ;GET THE DRIVE AND HEAD BYTE
    CALL   OUT_FDC    ;SEND IT TO THE FDC
    MOV AH,TRACK   ;GET THE TRACK NUMBER
    CALL   OUT_FDC    ;SEND IT TO THE FDC
    CALL   WAIT_INTERRUPT       ;WAIT FOR THE OPERATION TO COMPLETE

;WAIT FOR THE HEAD TO SETTLE

    MOV CX,1750    ;COUNT FOR EMPTY LOOP
WAIT_SETTLE:          ;IDLE FOR 25 MILLISECONDS
    LOOP   WAIT_SETTLE

;INITIALIZE THE DMA CHIP

    MOV AL,46H     ;OP-CODE TO READ DATA FROM THE FDC
    OUT 0BH,AL     ;WRITE TO THE MODE REGISTER
    OUT 0CH,AL     ;CLEAR FIRST/LAST FLIP-FLOP

;CALCULATE THE 20 BIT ADDRESS OF DATA_BUFF

    MOV AX,OFFSET DATA_BUFF ;GET BUFFER OFFSET IN DATA SEGMENT
    MOV BX,DS      ;PUT DS INTO BX
    MOV CL,4       ;GET READY TO ROTATE HIGH NIBBLE OF DS
    ROL BX,CL      ;ROTATE TO BOTTOM FOUR BITS OF BX
    MOV DL,BL      ;COPY BL TO DL
    AND DL,0FH     ;BLANK TOP NIBBLE OF DL
    AND BL,0F0H    ;BLANK BOTTOM NIBBLE OF BL
    ADD AX,BX      ;ADD BX INTO AX (DS INTO OFFSET)
    JNC NO_CARRY   ;IF NO CARRY, DL IS A PAGE VALUE
    INC DL     ;BUT IF CARRY, FIRST INCREMENT DL

;NOW SEND ADDRESS AND BYTE COUNT INFORMATION TO THE DMA CHIP

NO_CARRY:
    OUT 4,AL       ;SEND LOW BYTE OF ADDRESS
    MOV AL,AH      ;SHIFT HIGH BYTE
    OUT 4,AL       ;SEND HIGH BYTE OF ADDRESS
    MOV AL,DL      ;FETCH PAGE VALUE
    OUT 81H,AL     ;SEND  PAGE NUMBER
    MOV AX,4096    ;BYTE COUNT OF FULL TRACK OF DATA
    OUT 5,AL       ;SEND LOW BYTE OF DATA BYTE COUNT
    MOV AL,AH      ;READY HIGH BYTE
    OUT 5,AL       ;SEND HIGH BYTE OF DATA BYTE COUNT

;ENABLE CHANNEL 2 OF THE DMA CHIP

    MOV AL,2       ;GET SET TO ENABLE CHANNEL 2
    OUT 0AH,AL     ;ALL DONE, DMA WAITS FOR DATA

;NOW SEND THE READ PARAMETERS TO THE FDC

    MOV AH,062H    ;OP_CODE FOR FULL TRACK READ
    CALL   OUT_FDC    ;SEND IT TO THE FDC
    MOV AH,DRIVE_HEAD ;DRIVE AND HEAD BYTE, FROM SETUP
    CALL   OUT_FDC    ;SEND IT TO THE FDC
    MOV AH,TRACK   ;TRACK NUMBER, FROM SETUP
    CALL   OUT_FDC    ;SEND IT TO THE FDC
    MOV AH,HEAD    ;HEAD, FROM SETUP
    CALL   OUT_FDC    ;SEND IT TO THE FDC
    MOV AH,1       ;START READING AT SECTOR ONE
    CALL   OUT_FDC    ;SEND IT TO THE FDC
    MOV AH,SECTOR_SIZE          ;256 BYTE SECTORS FOR THE DIAGNOSTIC
DISK
    CALL   OUT_FDC    ;SEND IT TO THE FDC
    MOV AH,LAST_SECTOR          ;READ 16 SECTORS FROM THE TRACK
    CALL   OUT_FDC    ;SEND IT TO THE FDC
    MOV AH,GAP3_LENGTH          ;GAP3 LENGTH IN BYTES
    CALL   OUT_FDC    ;SEND IT TO THE FDC
    MOV AH,DATA_LENGTH          ;OFFH, READ ALL DATA IN THE SECTORS
    CALL   OUT_FDC    ;SEND IT TO THE FDC
    CALL   WAIT_INTERRUPT       ;WAIT FOR THE OPERATION TO COMPLETE

;TURN OFF THE MOTOR
```

Then the requisite series of bytes is sent. Some commands require as few as two bytes including the op-code. Others require as many as eight bytes after the op-code.

The program turns on the motor of a selected drive and goes into a delay loop to give the motor time to come up to speed. The drive is then recalibrated. Recalibration is one of the simple two byte commands. A byte containing the op-code 07H is sent followed by a byte containing a number representing the drive to be recalibrated. The FDC then goes into its execution phase and performs the commanded action. In this case the heads retract to track zero of the drive and the controller's track counter register is zeroed. In effect the drive and FDC have been reset. After an operation is completed the FDC returns an interrupt and drive status information to the processor interface.

Our program waits for the interrupt to be returned before proceeding. A seek command is now sent to the FDC, that is move the heads to the selected track. The op-code for a seek is 0FH, which is sent to the FDC followed by two additional bytes that carry the selected drive, head and track information. The FDC enters its execution phase, carries out the command and returns an interrupt and status information. A time delay is introduced at this point to allow the head to settle onto the spinning diskette. Data can now be read from the diskette. However, since we wish the data to be read from diskette to memory by DMA it is necessary that the program initialize the DMA chip before reading data from the diskette.

The IBM PC and compatibles use the 8237A programmable DMA controller chip. The 8237A has four independent prioritized transfer channels. Channel 0 has the highest priority, 0, and is used for memory refresh. Channel 1 is unassigned. Its priority is 1. Channel 2 is used for data transfer to and from the floppy disk drives. It has a priority of 2. Channel 3 is used for data transfer to and from the fixed disk. It has a priority of 3. The enclosed program concerns itself only with Channel 2, that assigned to the floppy disk drives.

Channel 2 of the DMA chip is directly addressed through six I/O ports. They are:

```
4    Used for the starting address of the memory area to or from which
     data will be transferred.
5    Used for the byte count of the data to be transferred.
0AH  Mask Register
0BH  Mode Register
0CH  Clear First/Last flip-flop
81H  Page Register
```

The bits of the Mode Register have the following meaning:

```
bits 0-1  Channel select, 00 = 0, 01 = 1, 10 = 2, 11 = 3
bits 2-3  Transfer: Verify = 00, Write = 01, Read = 10
bit  4    Autoinitialization, 0 = Disable, 1 = Enable
bit  5    Select Address, 0 = Increment, 1 = Decrement
bits 6-7  01 = Single mode select
```

The op-code is derived from the above bit meanings and is sent to I/O port 0BH to set the mode of DMA operation. In the enclosed program the op-code used is 046H. Which means Select Single Mode Transfer, Increment Addresses after each byte transfer, Disable Autoinitialization, Write to Memory, and Select Channel 2 to use the floppy disk drives.

A write to I/O port 0CH Clears the First/Last Flip/Flop. This command must be executed before writing a new address or new word count information to the DMA chip in order to initialize this flip/flop to a known state so that subsequent accesses to this register's contents by the microprocessor will address its upper and lower bytes in the correct sequence.

The purpose of I/O ports 4, 5 and 81H are self evident from the above. However, the manner of their use is not straight forward.

Since the IBM PC's address bus is twenty bits wide, the starting address of the memory area to which data read from the disk is to be transferred must be a twenty bit value. To handle this with

```
    MOV DX,03F2H  ;ADDRESS OF THE DIGITAL OUTPUT REGISTER
    MOV AL,MOTOR_OFF ;LEAVE BIT 3 ON, TURN BIT 2 OFF
    OUT DX,AL     ;SEND THE COMMAND TO THE FDC
    RET       ;RETURN TO CALLING ROUTINE

READ_TRACK    ENDP

WAIT_INTERRUPT   PROC          NEAR      ;WAITS FOR INTERRUPT 6, AND
RESETS STATUS BYTE
;MONITORS INTERRUPT 6 STATUS IN BIOS STATUS BYTE:

    MOV AX,40H    ;SEGMENT OF BIOS DATA AREA
    MOV ES,AX     ;PLACE IT IN ES
    MOV BX,03EH   ;OFFSET OF THE STATUS BYTE
WI1:   MOV DL,ES:[BX]          ;GET THE BYTE
    TEST  DL,80H    ;TEST BIT 7
    JZ WI1    ;KEEP LOOPING UNTIL SET
    AND DL,01111111B ;RESET BIT 7
    MOV ES:[BX],DL;REPLACE THE STATUS BYTE
    RET       ;RETURN TO CALLING ROUTINE
WAIT_INTERRUPT   ENDP

OUT_FDC PROC NEAR             ;SENDS THE BYTE IN AH TO THE FDC
    MOV DX,03F4H  ;FDC STATUS REGISTER PORT ADDRESS
KEEP_TRYING:
    IN AL,DX      ;FETCH STATUS REGISTER CONTENTS
    TEST  AL,80H    ;IS BIT 7 ON?
    JZ KEEP_TRYING  ;IF NOT, KEEP LOOPING
    INC DX     ;READY, POINT TO FDC DATA REGISTER PORT ADDRESS
    MOV AL,AH     ;VALUE WAS PASSED IN AH
    OUT DX,AL     ;SEND THE VALUE TO THE FDC DATA REGISTER
    RET       ;RETURN TO CALLING ROUTINE
OUT_FDC ENDP

;THE SHOW PROCEDURE DISPLAYS THE DATA THAT WAS READ INTO DATA_BUFF
;FROM THE SELECTED TRACK, ON THE VIDEO MONITOR
;THE DATA IS PRESENTED AS 16 SECTORS OF 256 CHARACTERS EACH
;EACH SECTOR IS LABELED WITH ITS TRACK, HEAD AND SECTOR NUMBER
;THE DATA IN EACH SECTOR IS SHOWN AS 4 LINES 64 CHARACTERS EACH

SHOW   PROC   NEAR
    MOV SI,OFFSET DATA_BUFF ;DATA_BUFF ADDRESS TO INDEX REGISTER
    MOV BL,1      ;BL IS SECTOR COUNTER
    MOV CX,10H    ;NUMBER OF SECTORS PER TRACK
SHOW1: PUSH   CX     ;SAVE CX AS SECTOR COUNTER
    MOV DX,OFFSET TRK_MSG   ;SETUP TO OUTPUT TRK_MSG
    CALL   MSGOUT     ;GO DO IT
    MOV DL,TRACK  ;OUTPUT TRACK NUMBER
    CALL   DECOUT     ;IN DECIMAL
    CALL   SPACE      ;OUTPUT A SPACE
    MOV DX,OFFSET HD_MSG    ;SETUP TO OUTPUT HD_MSG
    CALL   MSGOUT     ;GO DO IT
    MOV AL,HEAD   ;OUTPUT HEAD NUMBER
    ADD AL,30H    ;IN ASCII
    CALL   OUTPUT     ;GO DO IT
    CALL   SPACE      ;OUTPUT A SPACE
    MOV DX,OFFSET SEC_MSG   ;SETUP TO OUTPUT SEC_MSG
    CALL   MSGOUT     ;GO DO IT
    MOV DL,BL     ;OUTPUT THE SECTOR NUMBER
    CALL   DECOUT     ;IN DECIMAL
    CALL   CRLF       ;OUTPUT A CARRIAGE RETURN LINE FEED
    MOV CX,4      ;NUMBER OF LINES TO DISPLAY
SHOWA: PUSH   CX     ;SAVE CX AS LINE COUNTER
    MOV CX,40H    ;CX AS BYTE COUNTER
SHOW2: MOV DL,[SI]   ;OUTPUT THE CONTENTS OF THE BYTE
    MOV AL,DL     ;POINTED TO BY THE INDEX REGISTER
    CALL   OUTPUT
    CALL   KEY_CTRL  ;CHECK KEYBOARD FOR INPUT?
    INC SI     ;ADVANCE THE INDEX ADDRESS ONE BYTE
    LOOP   SHOW2      ;SHOW 64 BYTES
    CALL   CRLF       ;NEW LINE
    POP CX     ;GET CX AS THE LINE COUNTER
    LOOP   SHOWA      ;SHOW 4 LINES
    CALL   CRLF       ;NEW LINE
    INC BL     ;NEXT SECTOR NUMBER
    POP CX     ;GET CX AS SECTOR COUNTER
    LOOP   SHOW1      ;REPEAT FOR 16 SECTORS
SHOW4: RET        ;RETURN TO CALLING ROUTINE
SHOW   ENDP

;THE DECOUT PROCEDURE CONVERTS BINARY NUMBERS TO DECIMAL ASCII
NUMBERS
;THE INPUT IS A BINARY NUMBER IN REGISTER DL
;THE OUTPUT IS SENT TO THE VIDEO MONITOR FOR DISPLAY

DECOUT PROC   NEAR
    PUSHR         ;SAVE THE REGISTERS
    MOV CX,0      ;ZERO CX AS COUNTER
    MOV DI,OFFSET DECIMAL_BUFF ;DECIMAL_BUFF WILL RECEIVE THE
                  ;DECIMAL NUMBER
DECOUT1:
    PUSH   CX     ;SAVE THE COUNT
    MOV AL,DL     ;AX HAS THE NUMERATOR
    MOV AH,0      ;CLEAR THE UPPER HALF
    MOV CL,10     ;DIVISOR OF 10
    DIV CL     ;DIVIDE
    MOV DL,AL     ;PUT QUOTIENT IN DL
    MOV AL,AH     ;PUT THE REMAINDER IN AL
    ADD AL,30H    ;MAKE REMAINDER AN ASCII NUMBER
    MOV [DI],AL   ;STORE IT IN DECIMAL_BUFF
    INC DI     ;POINT TO NEXT BYTE OF DECIMAL_BUFF
    POP CX     ;GET THE COUNT
    INC CX     ;ADD ONE TO IT
    CMP DL,0      ;IS THE QUOTIENT ZERO?
    JNZ DECOUT1   ;NO, DIVIDE IT BY 10 AGAIN
DECOUT2:
    DEC DI     ;YES, BACK UP A BYTE IN DECIMAL_BUFF
```

eight bit I/O ports requires some manipulation. After the twenty bit address value has been computed the least significant eight bits are sent as a byte to I/O port 4, then the next significant eight bits are also sent as a byte to I/O port 4 and finally the most significant four bits, right justified in a byte, are sent to I/O port 81H where only the bits in the least significant nibble are used. The DMA controller is built to interpret the data it receives this way as a twenty bit address.

The byte count of the data to be transferred has a similar problem. Its value requires up to sixteen bits, a page of memory. The sixteen bits are sent to I/O port 5, least significant byte first followed by the most significant byte. Cross page transfers require a change in the value sent to the page register and therefore an update of the address and byte count values at each page boundary.

The channel number, in this case 2, is sent to I/O port 0AH, the Mask Register, to enable DMA for that channel. This is the last thing done in the initialization of DMA. The DMA chip now awaits the appearance of data, one byte at a time, at the Data Register of the FDC.

The program now instructs the FDC to read and send a full track of data to its Data Register one byte at a time. The "Read a Full Track" function of the FDC chip requires sending the chip a nine byte command string. The nine bytes contain the following information:

1. The op-code for "Read a Full Track", 062H
2. A byte with the drive and head information in it
3. The track number
4. The head number
5. The number of the sector at which to start reading
6. A code number indicating the size of the sectors
7. The number of the last sector on the track
8. The length of Gap 3 in bytes
9. The data length

Upon receipt of the ninth byte the FDC begins its read operation. When it has finished its operation the FDC will return an interrupt. Therefore, having started the read the program waits for the interrupt before moving on.

Before going further four bytes of the above FDC command string should be explained. Bits 0-1 of byte 2 carry the selected drive information, 00 = A, 01 = B, 10 = C and 11 = D. Bit 2 of byte 2 is the selected head information, 0 = one side, 1 = the other side. The remaining bits are filled with zeros. The short piece of code at the end of the SETUP procedure of the program forms this byte from the drive and head information input from the keyboard.

Bits 0-2 of byte 6 are used to inform the FDC of the number of bytes per sector on the diskette, for double-sided double-density diskettes, 001 = 256, 010 = 512, 011 = 1024, 100 = 2048 and 101 = 4096. The remaining bits are filled with zeros.

Bits 0-1 of byte 9 informs the FDC of the number of bytes to read from a sector, 0FFH is the normal value if byte 6 has one of the values given above. However, if byte 6 is made 0, the value in byte 9 is the number of bytes the FDC is to read from a sector. This permits the partial reading of sectors. Not a particularly useful function and therefore the value of byte 9 is usually made 0FFH which causes the FDC to read the number of byte per sector coded by byte 6 or complete sectors.

Floppy disk drives have errors in timing built into them which result from the fact that the devices are electro-mechanical in nature. No matter how carefully the parts are made there will be variations in the actions of the drives which will result in timing errors and therefore read/write errors. These errors are taken care of by the use of filler areas on the tracks called gaps.

The gaps are written on each track when the diskette is formatted. The tracks start with Gap 5, the pre-index gap. This gap starts at the physical index mark, the point in the diskette which is under

```
    MOV AL,[DI]    ;PUT CONTENTS OF THAT BYTE INTO AL
    CALL   OUTPUT     ;OUTPUT THE  DECIMAL DIGIT
    LOOP   DECOUT2    ;DO IT AGAIN TILL CX IS 0
    POPR           ;RESTORE THE REGISTERS
    RET        ;RETURN TO CALLING ROUTINE
DECOUT ENDP

;THE SETUP PROCEDURE DISPLAYS MESSAGES ON THE VIDEO MONITOR ASKING
FOR
;KEYBOARD INPUT OF THE DRIVE NUMBER, HEAD NUMBER AND TRACK NUMBER.
;IT CHECKS THE KEYBOARD INPUT FOR THE CORRECT RANGE OF VALUES IN EACH
;CASE AND STORES THE SELECTED VALUES IN THE VARIABLES DRIVE,
MOTOR_ON,
;HEAD AND TRACK.  FURTHER, IT CALCULATES AND STORES THE VALUE FOR
;DRIVE_HEAD.

SETUP  PROC   NEAR
GETDRV: CALL CRLF              ;NEW LINE
    MOV DX,OFFSET MSG_DRV  ;SETUP TO OUTPUT THE GET DRIVE MESSAGE
    CALL   MSGOUT     ;GO DO IT
    CALL   KEYIN      ;GET A CHARACTER FROM THE KEYBOARD
    CMP AL,0       ;IF IT IS A 0, YOU WANT DRIVE A
    JE  DRVA       ;GO TO DRIVE A SETUP ROUTINE
    CMP AL,1       ;IF IT IS A 1, YOU WANT DRIVE B
    JE  DRVB       ;GO TO DRIVE B SETUP ROUTINE
    CALL   ERROR      ;ANY OTHER CHARACTER IS AN ERROR
    JMP GETDRV     ;SAY SO AND REPEAT GETDRV

DRVA:  MOV DRIVE,0    ;PUT A 0 INTO THE VARIABLE DRIVE
    MOV MOTOR_ON,1CH  ;INIT MOTOR_ON TO TURN DRIVE A ON
    MOV MOTOR_OFF,0   ;INIT MOTOR_OFF TO TURN DRIVE A OFF
    JMP GETHD      ;GO GET THE HEAD NUMBER
DRVB:  MOV DRIVE,1    ;PUT A 1 INTO THE VARIABLE DRIVE
    MOV MOTOR_ON,2DH  ;INIT MOTOR_ON TO TURN DRIVE B ON
    MOV MOTOR_OFF,1   ;INIT MOTOR_OFF TO TURN DRIVE B OFF

GETHD: MOV DX,OFFSET MSG_HD    ;SETUP TO OUTPUT GET HEAD MESSAGE
    CALL   MSGOUT     ;GO DO IT
    CALL   KEYIN      ;GET CHARACTER FROM THE KEYBOARD
    CMP AL,0       ;WAS IT 0?
    JZ  SETHD      ;YES, SET HEAD TO 0
    CMP AL,1       ;NO, WAS IT 1?
    JZ  SETHD      ;YES, SET HEAD TO 1
    CALL   ERROR      ;ANYTHING ELSE IS AN ERROR, SAY SO
    JMP GETHD      ;AND REPEAT GETHD
SETHD: MOV HEAD,AL    ;STORE HEAD NUMBER IN VARIABLE HEAD

GETTRK: MOV   DX,OFFSET MSG_TRK  ;SETUP TO OUTPUT GET TRACK MESSAGE
    CALL   MSGOUT     ;GO DO IT
    MOV DX,0       ;CLEAR DX
GETDIG: MOV   AH,1             ;GET A CHARACTER FROM THE KEYBOARD
    INT 21H    ;BY CALL TO DOS INTERRUPT
    CMP AL,0DH     ;WAS IT AN ENTER (CARRIAGE RETURN)
    JE  SETTRK     ;YES, YOU HAVE IT ALL, GO SET THE TRACK
    CMP AL,'0'           ;IF INPUT CHARACTER IS LESS THAN '0' ASCII
    JB  NOGOOD     ;DISALLOW IT
    CMP AL,'9'           ;IF INPUT CHARACTER IS MORE THAN '9' ASCII
    JA  NOGOOD     ;DISALLOW IT
    AND AL,0FH     ;CONVERT ASCII NUMBER TO BINARY
    CBW        ;CONVERT BYTE TO WORD ( AX = AL )
    PUSH   AX      ;SAVE THE NUMBER
    MOV AX,DX      ;AX IS DX
    MOV CX,0AH     ;PREPARE TO MULTIPLY AX BY 10
    MUL CX      ;MULTIPLY
    MOV DX,AX      ;SAVE THE PRODUCT IN DX
    POP AX      ;GET THE NUMBER THAT WAS SAVED IN AX
    ADD DX,AX      ;ADD IT TO WHAT IS IN DX
    JMP GETDIG     ;GET THE NEXT DIGIT FROM THE KEYBOARD

NOGOOD: CALL ERROR             ;OUTPUT THE ERROR MESSAGE AND
    JMP GETTRK     ;REPEAT THE GET TRACK ROUTINE

SETTRK: CMP   DL,0             ;DL HAS THE TRACK NUMBER IN IT
    JB  NOGOOD     ;IF THE TRACK NUMBER IS LESS THAN 0 REDO IT
    CMP DL,27H     ;IF THE TRACK NUMBER IS MORE THAN 39
    JA  NOGOOD     ;REDO IT
    MOV TRACK,DL   ;OK, STORE TRACK NUMBER IN VARIABLE TRACK
    CALL   CRLF       ;OUTPUT A NEW LINE

;MAKE UP THE DRIVE_HEAD BYTE FROM THE INPUT VALUES OF DRIVE AND HEAD

    MOV DL,DRIVE   ;DRIVE IS IN BITS 0-1 OF DL
    MOV AH,HEAD    ;HEAD IS IN BIT 0 OF AH
    SAL AH,1       ;SHIFT HEAD BIT TO BIT 1 OF AH
    SAL AH,1       ;SHIFT HEAD BIT TO BIT 2 OF AH
    AND AH,4       ;ZERO ALL BITS OF AH BUT BIT 2
    OR  AH,DL      ;PUT DRIVE BITS INTO BITS 0-1 OF AH
    MOV DRIVE_HEAD,AH ;STORE VALUE IN DRIVE_HEAD BYTE
    RET        ;RETURN TO CALLING ROUTINE

ERROR: MOV DX,OFFSET ERR_MSG ;SET-UP TO OUTPUT THE ERROR MESSAGE
    CALL   MSGOUT     ;GO DO IT
    RET        ;RETURN TO CALLING ROUTINE

SETUP  ENDP

;KEYIN (KEYBOARD INPUT) PROCEDURE USES A DOS INTERRUPT FUNCTION
;TO INPUT A SINGLE CHARACTER FROM THE KEYBOARD.  THIS ROUTINE IS
;USED TO INPUT DIGITS FROM THE KEYBOARD AND CONVERT THE ASCII
;FORM TO BINARY FOR USE BY THE PROGRAM.  THE BINARY BYTE IS
;RETURNED IN REGISTER AL.

KEYIN  PROC   NEAR
    MOV AH,1       ;USE DOS INTERRUPT CALL TO INPUT
    INT 21H    ;A DIGIT FROM THE KEYBOARD
    SUB AL,30H     ;CONVERT FROM ASCII TO BINARY
    CALL   CRLF       ;OUTPUT A CARRIAGE RETURN LINE FEED
```

the heads when the small hole in the diskette passes its LED and photo cell detector. Gap 5 is followed by the index address mark which identifies the start of a track. This in turn is followed by Gap 1, the post index gap.

Next is written sector information for sector 1. It starts with a Data Address Mark followed by the ID field for sector 1. This contains track, head, sector address, sector length and two bytes for CRC information for the sector. Then follows the post sector ID field Gap 2. Then the sector 1 data field is written filled with a fill character supplied by the formatting program. This field is followed by the post data field Gap 3.

The sector information above is repeated with only a change in the sector number for however many sectors are to be written to the track. The final gap written on the track is Gap 4 which is a filler used to take up the space between the last physical data field on the track and the physical index mark, the start of Gap 5, the start of the track.

The size in bytes of Gap 1, Gap 2, Gap 5, the Index Address Mark and the sector ID fields are fixed and are written by the FDC during formatting. To allow for different track formats Gap 3 has been made variable and its length can be selected by the programmer. Its minimum size must be long enough to include the write turn-off time of the drive as Gap 3 passes under the write head. Its maximum size must be small enough to permit all sectors to fit on to the track. Gap 3 for the last sector on the track becomes part of Gap 4. The length of Gap 4 is also variable but not under direct control of the programmer, as is Gap 3. Its length depends upon the format selected for the diskette. The control electronics of the drive computes its length such that the total number of bytes recorded on the formatted diskette will equal the nominal unformatted capacity of the track which is 6250 bytes for double-density recording. There are of course recommended values for Gap 3 for standard numbers of sectors on a track. The value in byte 8 of the command string is the length of Gap 3 in bytes which will properly read the 16 sector 256 byte per sector format used by Dysan for its double-sided double-density Digital Diagnostic Diskette.

When the interrupt occurs indicating that the read operation has been completed our program turns off the motor of the selected drive.

The remaining procedures in the program are straight forward coding. The one called SHOW displays on the video monitor the data that has been read from the disk by the drive. It presents the data a sector at a time in four lines of sixty four characters each. Each sector is labeled with its track number, head number and sector number. The DECOUT procedure converts binary numbers to decimal ASCII numbers for use in the labels.

The SETUP procedure is the part of the program that interacts with the user. It asks for the drive number, the head number and the track number, checks their values to assure that they are within proper range, asks for corrections if they are not and fills in the variables that are needed by the READ-TRACK routine.

By the use of this program and the Dysan Diagnostic Diskette I was able to determine that my drives were indeed in need of alignment. These were QumeTrak 142 drives. As received the best they would do was read out to sectors seven and eight on the progressive offset tracks. I was able to realign them by taking the tension off the mounting screws that hold the stepper motor in place, then using a prick punch and a light hammer I gently rapped the mounting ears and ran the program to see what progress had been made. In a very few tries I was able to reposition the stepper motors so that both drives would read out to sectors eleven and twelve. Then I retensioned the mounting screws and tried to run Flight Simulator. It loaded and ran from either drive. I quit while I was ahead. I have had no further problem with these drives.

Other tests can be run with this program since it will read other tracks on the Digital Diagnostic Disk in addition to the progressive offset ones. For example, I ran the Head Azimuth Alignment test

```
    RET         ;RETURN TO CALLING ROUTINE
KEYIN  ENDP

;MSGOUT (MESSAGE OUTPUT) PROCEDURE USES A DOS INTERRUPT FUNCTION TO
;OUTPUT AN ASCII STRING TO THE VIDEO DISPLAY.  THE STARTING ADDRESS,
;ITS OFFSET IN THE DATA SEGMENT, MUST BE IN REGISTER DX WHEN THE
;PROCEDURE IS CALLED.  THE ASCII STRING MUST BE TERMINATED BY THE
;CHARACTER $.

MSGOUT PROC NEAR
    MOV AH,9
    INT 21H
    RET         ;RETURN TO CALLING ROUTINE
MSGOUT ENDP

;KEY_CTRL PROCEDURE IS DESIGNED TO GIVE THE OPERATOR CONTROL OF THE
;PROGRAM DURING THE DISPLAY OF WHAT HAS BEEN READ FROM THE SELECTED
;TRACK.
;   THREE INPUTS HAVE MEANING
;   CONTROL C = QUIT AND RETURN TO DOS
;   CONTROL A = INTERRUPT AND RESTART THE PROGRAM ANEW
;   ESC   = A TOGGLE THAT STOPS AND STARTS THE DISPLAY
;
;ANY OTHER INPUT IS IGNORED.
;
;ALL REGISTERS ARE SAVED AND RESTORED BY THE PROCEDURE.

KEY_CTRL PROC  NEAR
    PUSHR           ;SAVE THE REGISTERS
    MOV AH,6        ;PICK-UP KEYBOARD INPUT IF THERE IS ANY
    MOV DL,0FFH     ;BUT DO NOT WAIT FOR IT
    INT 21H     ;RETURN TO CALLING PROGRAM IF
    JZ  KEY4        ;THERE IS NO INPUT
    CMP AL,3        ;IF THERE WAS INPUT, WAS IT A CTRL_C?
    JNE KEY1        ;NO, MAKE THE NEXT TEST
    MOV AH,4CH      ;RETURN TO DOS
    INT 21H     ;BY CALL TO DOS INTERRUPT
KEY1:  CMP AL,1         ;WAS THE INPUT A CTRL_A?
    JNE KEY2        ;NO, MAKE THE NEXT TEST
    JMP START       ;YES, RESTART THE PROGRAM
KEY2:  CMP AL,1BH       ;WAS THE INPUT AN ESC?
    JNE KEY4        ;NO, RETURN TO CALLING ROUTINE
KEY3:  MOV AH,8         ;YES, HALT THE DISPLAY AND WAIT FOR MORE
    INT 21H     ;KEYBOARD INPUT
    CMP AL,1BH      ;HAS ANOTHER ESC COME IN?
    JNE KEY3        ;NO, LOOP UNTIL IT DOES
KEY4:  POPR             ;RESTORE THE REGISTERS
    RET         ;RETURN TO CALLING ROUTINE
KEY_CTRL ENDP

;DELAY PROCEDURE TAKES UP TIME, IN DO NOTHING LOOPS, PROPORTIONAL
;TO THE PRODUCT OF THE INITIAL VALUES IN THE BX AND CX REGISTERS.
;NO INPUT IS REQUIRED.
;
;ALL REGISTERS ARE SAVED AND RESTORED BY THE PROCEDURE.

DELAY  PROC   NEAR
    PUSHR           ;SAVE THE REGISTERS
    MOV CX,1000     ;USE CX AS A COUNTER
    MOV BX,600      ;USE BX AS ANOTHER COUNTER
HOLDIT: LOOP  HOLDIT            ;NULL LOOP TILL CX = 0
    MOV CX,1000     ;RESET CX
    DEC BX      ;REDUCE BX BY ONE
    JNE HOLDIT      ;DO THE NULL LOOPS AGAIN
    POPR            ;RESTORE THE REGISTERS
    RET         ;RETURN TO CALLING ROUTINE
DELAY  ENDP

;CLRSCR (CLEAR SCREEN) PROCEDURE USES A DOS INTERRUPT FUNTION TO
;CAUSE THE SCREEN OF THE VIDEO DISPLAY TO BE CLEARED AND TO
;HOME THE CURSOR.  NO INPUT IS REQUIRED.
;
;ALL REGISTERS ARE SAVED AND RESTORED BY THE PROCEDURE.

CLRSCR PROC NEAR
    PUSHR           ;SAVE THE REGISTERS
    MOV AX,2        ;CALL DOS TO CLEAR THE SCREEN
    INT 10H     ;AND HOME THE CURSOR
    POPR            ;RESTORE THE REGISTERS
    RET         ;RETURN TO CALLING ROUTINE
CLRSCR ENDP

;CRLF PROCEDURE (NEW LINE) PLACES A CARRIAGE RETURN CHARACTER IN THE
;REGISTER AL AND CALLS OUTPUT TO TRANSFER IT TO THE VIDEO DISPLAY.
;THE PROCEDURE THEN DOES THE SAME WITH A LINE FEED CHARACTER.
;NO INPUT IS REQUIRED.
;
;ALL REGISTERS ARE SAVED AND RESTORED BY THE PROCEDURE.

CRLF PROC NEAR
    PUSHR           ;SAVE THE REGISTERS
    MOV AL,0DH      ;OUTPUT A CARRIAGE RETURN
    CALL   OUTPUT
    MOV AL,0AH      ;OUTPUT A LINE FEED
    CALL   OUTPUT
    POPR            ;RESTORE THE REGISTERS
    RET         ;RETURN TO CALLING ROUTINE
CRLF ENDP

;SPACE PROCEDURE PLACES THE CHARACTER ' ', (SPACE) IN REGISTER AL
;AND CALLS OUTPUT TO TRANSFER SPACE TO THE VIDEO DISPLAY.
;
;ALL REGISTERS ARE SAVED AND RESTORED BY THE PROCEDURE.

SPACE PROC NEAR
    PUSHR           ;SAVE THE REGISTERS
    MOV AL,' '          ;OUTPUT A SPACE
```

by reading track 34. However, correcting a problem with Head Azimuth Alignment is not field serviceable for QumeTrak 142 drives. My drives appeared to be pretty good. However, I could have done nothing about it if they had not.

Modifications to the program would be required to run the other tests listed above. However, not having the specifications the drives should meet would make the test results all but meaningless.

If you are going to buy and use drives of unknown condition and history I strongly recommend you also buy a Dysan Digital Diagnostic Diskette so as to be able to test and adjust the Head Radial Alignment for yourself. Being able to do so and also to test and adjust drive rotational speed with available public domain software will enable you to standardize your drives and keep them that way. •

```
    CALL   OUTPUT
    POPR          ;RESTORE THE REGISTERS
    RET        ;RETURN TO CALLING ROUTINE
SPACE ENDP

;OUTPUT PROCEDURE USES A DOS INTERRUPT FUNCTION TO OUTPUT A CHARACTER
;TO THE VIDEO DISPLAY.        THE PROCEDURE REQUIRES THAT THE
CHARACTER TO
;BE OUTPUT BE IN REGISTER AL WHEN THE PROCEDURE IS CALLED.
;
;ALL REGISTERS ARE SAVED AND RESTORED BY THE PROCEDURE.

OUTPUT PROC NEAR
    PUSHR          ;SAVE THE REGISTERS
    MOV DL,AL      ;PLACE CHARACTER TO BE OUTPUT IN DL
    MOV AH,2       ;USE DOS INTERRUPT CALL TO OUTPUT
    INT 21H     ;THE CHARACTER
    POPR           ;RESTORE THE REGISTERS
    RET        ;RETURN TO CALLING ROUTINE
OUTPUT ENDP

CODE ENDS

    END MAIN
```

References

Testing Your Disk Drive, Mini-Micro Systems for June 1981 Cahners Publishing Company.

Digital Diagnostic Diskette - Information Dysan CE Division—A Xide Company.

Loren Amelang *Dysan's Digital Diskette Diagnostic* Dr. Dobb's Journal, December 1983.

Service Manual for the *Qumetrk 142 Flexible Disk Drive.*

Microsystems Components Handbook Volumes 1 & 2, INTEL 1985.

Jeffrey P. Royer *Handbook of Software & Hardware Interfacing for the IBM PCs,* Prentice-Hall 1987.

Robert Jourdain *Programmer's Problem Solver for the IBM PC, XT & AT,* Brady Communications Company, Inc. 1986.

User Disk

The code from *Standardize Your Floppy Disk Drives* and *Graphics Programming With C On the IBM PC* are available on a 5.25" 360K PC format disk for $10 postpaid in the US.

Xerox 16 / 8 DEM-II Computers

New dual system computers with the Disk Expansion Module. These systems include the following:

- Z80A 4 MHz CPU with 64 K of RAM
- 8086 4.77 MHz CPU with 128 K RAM
- 2 Serial ports
- 1 Parallel port
- 10 Meg 5.25" hard drive (**NOT** 8")
- 322 K DSDD floppy drive
- Low-profile programmable keyboard
- Monitor

CP/M-80 2.2, CP/M-86, and "Select" word processor are included. MS-DOS 2.01 is available as an option for an additional $35.

Cost is $329 plus $50 shipping in the US. This also includes a one year subscription to The Computer Journal (current subscribers should include a photocopy of their label so that their subscription can be extended). Registered owners of NZCOM receive a discount of $15. If you order NZCOM **at the time of the order,** deduct the $15. Order by personal check, bank cashier's check or money order. Personal checks held ten days. Allow 4 to 6 weeks for delivery.

Chris McEwen – Socrates Z Node 32
PO Box 12, S. Plainfield, NJ 07080
(201) 754-9067 3/12/24 bps

LSH

A New History Shell for Z-System

by Rob Friefeld

Scripts, aliases, shells—a number of command line generating aides were built into Z-System from the beginning. Over the past few years, more and more powerful tools have been developed to make use of them. In this article, I would like to describe a history shell called LSH.

What is a History Shell?

A Z-System shell is a program which runs instead of the ZCPR3 command processor whenever you would normally see the system prompt. The shell then accepts input in its own fashion. What distinguishes a shell from a regular utility is that the shell can turn control over to other programs and be automatically reinvoked when they are finished--much like ZCPR itself. The effect is to extend the command processor by surrounding it with a "shell" for the user's convenience.

A simple use of this wondrous process would be a front end program to extend the rudimentary line editing capabilities of the command processor (essentially, "delete last character" and "start over"). It would be nice to have text editing features, such as cursor movement and character insertion, to correct mistakes. Once the edited line is complete, the shell loads it into the command line buffer and turns it over to ZCPR3 for execution, just as if it had been entered directly at the ZCPR3 prompt. If we add to that a mechanism for recording and recalling command lines, we have the makings of a history shell.

Here are some history shell uses and their benefits that immediately come to mind:

```
1) Reuse command lines.
   - Save a lot of typing.
   - Avoid making typos.
   - Write more complex and efficient command lines.
2) Edit command lines.
   - Correct typing errors on the current line.
   - Correct commands which didn't work.
   - Make alterations before re-running a command.
   e.g. LINK TEST/N /A:100 TEST,Z3LIB/S,SYSLIB/S ,/E
      LINK TEST/N /A:8000 ... etc.
3) Keep a command record.
   - What commands did I use the last time I did this?
   - How did I get into this mess?
```

Those who type commands more than once would likely be delighted with these features. There is a price to pay, though. System response time is slowed by extra disk accesses. The history shell must load again after every command line is run, even the most trivial. It must also update the history. There is, then, some limit on the size of a practical history shell. On a floppy based system, history shells perform with all the agility of a walrus on roller skates. Hard disk performance is quite acceptable, and a RAM disk is wonderful.

A history shell could be implemented as a memory resident system extension (RSX) or an RCP segment. With no disk access, it would run instantly. (Carson Wilson's new Z34RCP 1.1 contains such a shell.) The price there is a reduction of free memory, an absolute minimum of program features, and some awkwardness in recording the history permanently.

History Shell Background

LSH was developed out of experience with three predecessors: Michael Rubenstein's HSH, NHSH by John Poplett (a conversion of HSH from C to assembler with additional features), and Paul Pomerleau's EASE. The latter is the most powerful of these.

In brief, the EASE user interface appears to be a blank line, as in the normal operating environment. You can type a line and execute it in the usual way. It is then appended to a history file consisting of null terminated command lines, as many as will fit on disk. Old lines can be recalled in reverse sequence or by match to a few starting characters. Configuration possibilities include complete customization of the command key set, and not recording short or recalled lines. A separate maintenance utility can be run on the history file now and then to clean out duplicate lines and trim down its size.

LSH History Shell

LSH shares many of these features but has at heart a different paradigm. LSH may be thought of as a full screen text editor which generates command lines. Its history file is an ordinary text file named on invocation. Command lines are recalled by moving the cursor back in the file, matching a string, or setting a pointer to repeat a command sequence. Because the output file is text, it is accessible to other text handling tools. For example, a full-size editor can be brought to bear for duties such as block deletes, block moves, or global find/replace.

Recording the history to a text file does not in itself require a screen oriented editor. In fact, LSH has a line mode which superficially appears to work like the regular CPR. However, the full screen editor interface literally adds another dimension to the history shell idea. Once a few lines are typed, we have what amounts to an instant menu of command lines to chose from. Just move the cursor to one of them, and press CR (carriage return). When we consider an array of command lines as TEXT, it is easy to conceive of concatenating them, splitting them up, shuffling their order, introducing explanatory comments, deleting useless ones, etc. In short, we have much more control over the history being generated.

I'll admit that LSH is not producing a true history, since history cannot be changed. LSH's "meta-history" is in reality a command file, hence the default history file type CMD. Rather than faithfully recording every command line—good, bad, or ugly—the LSH history file evolves as we refine it to help handle the tasks at hand. EASE's history is more like a ticker tape: you can only update it at the end.

Invoking LSH

The history file name can be specified on the command line, e.g.,

```
C15:WORK>LSH PROJECT
```

If no directory is specified for the history file, LSH first looks for the named file in the logged directory, then in the directory from which LSH.COM was loaded. For example, if LSH resides on a RAM disk directory named TOP, which is on the path, then the example command line starts LSH with WORK:PROJECT.CMD, or, if that file does not exist, TOP:PROJECT.CMD. If that file also does not exist, then it will be created. (The ZSDOS path feature adds still other possibilities.)

If no log file name is given, LSH uses a default file name--LSH.CMD. You may wish to rename LSH, in which case the default file name is taken from the current program name, e.g., X.CMD. The default file type can be installed if you prefer VAR or something else, or can be set blank. If your LSH command line specifies a file with an explicit file type, then it will be used.

Using LSH

Whenever carriage return is pressed, the current line (the one where the cursor is) is put into the command line buffer and run. (Lines headed with ";" are comments and do not run.) If you do nothing but type command lines and run them, LSH does nothing but record them; its presence won't be that obvious. When you switch to Screen Mode with ESC ESC (pressing the escape key twice), you will see something like this:

```
C15:WORK  10:30p  +
----------------------
zde mxo-sb31.z80
slr mxo-sb31/h
mload mex.com,mxo-sb31.hex
mex
--------------------
```

The top line shows the logged DU:DIR and the system time. The "+" indicates that recording is ON. A "-" appears when it is OFF. (The Line Mode prompt shows ">>".)

The number of lines in the text window may be set on-the-fly (ESC T). I find that a window of 4-5 commands is enough to tell where I am in the file and what is coming. Performance is also crisper when the entire screen doesn't have to be rewritten. To really get a look at the history, though, the entire screen can be used.

None of the usual "save and exit" or "save and continue" commands are implemented. Instead, LSH has a recording toggle (ESC S, Save). When ON, the history file is updated on every command execution if there have been any changes to it. When OFF, you can still enter, edit, and execute lines, but any changes won't be there when LSH is back. To get out of the shell, use ESC Q (Quit).

Normally, LSH leaves its display at the top of the screen and console output appears below. If LSH is using many lines, you may need a shell pause to examine the screen before LSH resumes. For example, the output of a DIR command could be obliterated before there is a chance to look at it. You can force a pause by pressing ESC-CR instead of CR to execute a line. For convenience, LSH automatically does a shell pause when it is set to show 10 or more history lines. The 10 line decision point may be configured to 0 for a pause every time, or to 99 for no pauses.

One other point on shell operation: as LSH is distributed, ^C is used as an editing control in Screen Mode. Initiating a warm boot requires ESC C.

Using the Editor

I won't go into detail on the usual editing functions. The default command key set uses the WordStar standard as a starting point. It may be completely reconfigured with the installation program, LSHINST.

LSH has two text recovery functions. "Unkill" (^U) is for recovering text deleted by a "delete line" or "delete to end of line" command. This is not only for recovering from a mistake, but for moving a string to a different cursor position. "Undo" (^QU) cancels any changes made to the current line UNLESS it has been deleted. (It works like the undo command in the Turbo Pascal editor.) The ultimate "undo" is to toggle recording OFF then execute a blank command line. That recovers the history unchanged.

There is one mass deletion command, "zap" (^KZ). It clears everything after the cursor position.

LSH has three methods of jumping to a specific point in the history file: string search, line sequencing, and position markers.

String Search

After the manner of HSH, LSH has a simple string search feature. Type a few characters, then initiate the search (^O, Old Line). LSH looks back for another line which begins with the same string. When found, the line is displayed without changing the cursor position. Another search command will then automatically look for the same string farther back in the history. For example, after the series of command lines:

```
1   ZDE LSH.ART
2   CP LSH.ART BACKUP:
3   ZDE LSH.LOG
```

the cursor will be on the beginning of a fourth, blank line. Typing "ZDE^O" moves the cursor to line 3. Another ^O (without moving the cursor) goes to line 1.

When the search is successful, the line the search started on is not changed. That means that the search string can be typed on a blank line, or even at the start of a non-blank line, without permanently entering anything. In the example, line 4 will still be blank when we return to the end of the history.

The search direction may be toggled forward or backward (ESC O). The current direction is saved on the shell stack between LSH runs.

Automatic Line Sequencing

Certainly the main idea of a history shell is that old commands may be reused. More than that, we will often want to repeat a sequence of command lines. The history file I used to work on LSH has sequences of this sort:

```
....
ZDE LSH.Z80
SLR LSH/R
ZDE LEDIT.Z80
SLR LEDIT.Z80
LINK LSH/N /A:8000 /J LSH,LEDIT,HRMIN,BGSIG,/E
....
```

You can imagine how many times I cycled through those commands! To facilitate that, LSH has an "auto line sequencing" toggle (ESC L). When OFF, LSH comes up on a blank line at the end of the history file. A "line recall" command (^L) jumps to the line following the one just executed. When ON, a line recall is done automatically, presenting each line of the history in turn. A simple carriage return continues the sequence.

For convenience, the screen and line modes use independent flags for this function. You can set screen mode to track the sequence while line mode comes up at the end of the history.

The feature works by saving a pointer to the text line wanted. It usually works well but can be fooled, such as when the history is edited independently or when the recording toggle is OFF but the text has been changed.

Markers

A third way to jump about in the history file is by setting line markers. To mark the current line, press ^PZ. Any number of markers may be set. To go to a mark, press ^QZ. All the marked lines are visited in reverse order to the beginning of the history, then wrapped to the end. To clear all of the markers, use ^KH. Use a marker in conjunction with line sequencing to cycle a series of lines repeatedly.

Advanced Features—The Queue

By an "advanced feature", I mean one that you will forget

about if you don't use LSH regularly—and that you can lead a perfectly happy life without.

I didn't want to greatly increase the size of LSH to include the usual block operations of a text editor. They wouldn't be used that often, and in any case, we can just call up our editor for such service. Nevertheless, I thought it would be handy to have some internal way to move command lines around. A queue was easy to implement and also has some advantages in this application.

The available queue commands (with default control settings) are:

```
^KB   Add line to queue, advance to next line.
^KK   ''Keep and Kill'' (add to queue, delete from history).
^KI   Insert the lines from the queue at the cursor line.
^KV   Do a ^KI, then clear the queue.
^KY   Just clear the queue.
^B    Duplicate the current line.
```

With a queue, we quickly pick up lines in the order we want them, then deposit them elsewhere or save them from a mass deletion. It is much easier to reorder a list with a queue than with WordStar-style block commands. Try it!

The line duplication command is a one-keystroke way to get another copy of the current line. It may be installed to automatically substitute the string "GO" for the first token.

Advanced Features—Token Repeat

Token repeat is a typing aid. Tokens are considered to be words separated by spaces or certain other punctuation. The default separators are ;:.,/=. Consider the line:

```
ZDE MYPROG.Z80;Z80ASM MYPROG/R;LINK MYPROG/N MYPROG /E
```

The "MYPROG" token does not have to be retyped each time. Just use the editing command to repeat token #2 (TAB 2). The token will be entered at the cursor according to the setting of the insert/overwrite toggle. Token repeat works for token numbers 1 - 9, but it would take a sharp eye to count beyond the first few.

Token repeat puts the selected token in the same buffer used by the unkill command. Therefore, an unkill (^U) will enter the last selected token again, even if the cursor is now on another line.

There is one other wrinkle to this feature. The text from the cursor to the end of the line can be put into the unkill buffer directly (TAB TAB). I frequently use this to replicate and re-run just the end of a multiple command line.

Log File Size

There wouldn't be much point to a full screen editor if only a line or two were in memory at one time. On the other hand, reading in a big history file on every invocation takes time and memory. As a compromise, LSH reads and writes the entire file on each invocation until it reaches a preset size, say 5k. After that, LSH only has access to the last 5k of the file. The history file continues to grow without taking up more and more memory and without progressively slowing system response. The default 5k size would hold several hundred typical command lines.

The preset size may be installed with LSH's configuration program. The type-4 version of LSH automatically adjusts its run location to the chosen buffer size. The type-3 version always runs at 8000h and may give an overflow message if there is not enough room in the TPA to handle a large file buffer. Most systems will have room for 8-12k of history file.

On-Line Help

HELPLSH.COM is a separate utility which displays a help screen of control key bindings as currently installed. It obtains its information from LSH's memory image, so it must be run from within LSH. I chose not to include the code for the help screen in LSH itself to keep the program size down.

"HELPLSH" is actually an internal command line which is run automatically when the help key is pressed (ESC J). A benefit to practiced users is that this internal command line may be set to something else. For example, it could be set to run an ARUNZ alias (e.g. /LSHCMD) which in turn could be set up for whatever job seemed useful at the moment. I use ZFILER frequently and have simply installed the line as "ROOT:ZF".

LSH Installation Program

One of the major design goals was to make LSH as easy as possible to completely reconfigure. I am happy using a control key set similar to WordStar's, but many of you strongly prefer something else. With about four dozen commands and numerous options, hand patching or editing an overlay was out of the question. LSH has a menu-oriented installation program which makes it convenient to alter things to your taste. But LSH doesn't NEED to be installed to run.

Companion Error Handler

Many Z-System programs invoke an error handler program when a command line cannot be processed for some reason, such as a reference to a nonexistent file or directory. The error handler describes the problem and presents the bad command for disposition. ZERRLSH is one such error handler which additionally checks to see if LSH is the current shell. If so, it will automatically enter any corrections you make to a command line into the history file.

In Conclusion

Ideas and suggestions still appear, so LSH is still under development. As I write this, the current version of LSH is 1.0r. It is available for download on Los Angeles' Ladera Z-Node (Z-Node Central) at 213-670-9465. The program library contains a type-3 version which runs at 8000h, a type-4 version, LSHINST.COM (for the installation), LSH.WS (the documentation), and HELPLSH.COM. You can also pick up ZERR12.LBR containing the error handler for use with LSH.

If you haven't tried a history shell yet, I think there is a treat in store for you. •

Registered Trademarks

It is easy to get in the habit of using company trademarks as generic terms, but these trademarks are the property of the respective companies. It is important to acknowledge these trademarks as their property to avoid their losing the rights and the term becoming public property. The following frequently used trademarks are acknowledged, and we apologize for any we have overlooked.

Apple II, II+, IIc, IIe, Lisa, Macintosh, DOS 3.3, ProDos; Apple Computer Company. CP/M, DDT, ASM, STAT, PIP; Digital Research. DateStamper, BackGrounder ii, Dos Disk; Plu*Perfect Systems. Clipper, Nantucket; Nantucket, Inc. dBase, dBASE II, dBASE III, dBASE III Plus, dBASE IV; Ashton-Tate, Inc. MBASIC, MS-DOS, Windows, Word; MicroSoft. WordStar; MicroPro International. IBM-PC, XT, and AT, PC-DOS; IBM Corporation. Z80, Z280; Zilog Corporation. Turbo Pascal, Turbo C, Paradox; Borland International. HD64180; Hitachi America, Ltd. SB180; Micromint, Inc.

Where these and other terms are used in The Computer Journal, they are acknowledged to be the property of the respective companies even if not specifically acknowledged in each occurrence.

Letwin's Prior Progeny

Heath's HDOS, Then and Now

by Kirk L. Thompson, Editor, The Staunch 8/89'er

From a purely "logical" point of view, the set of operating systems programmers appears to be quite small. Certain names crop up almost all the time and one of these is J. Gordon Letwin. He is best know today as chief architect of MicroSoft's OS/2. But among aficionados of Heath/Zenith's 8-bit systems, he is better know as the original developer of Heath's Disk Operating System (HDOS).

This DOS never became a threat to CP/M's dominance as the standard OS for 8-bit equipment, mainly because of the specific hardware it was designed to run on. But there has been some exciting activity over the last five years, some of it mirroring that of those enterprising CP/M programmers who have developed ZCPR, NZ-COM and ZSDOS. On the other hand, Heath Co. has provided more of its traditional support than its customers have reason to expect. I shall discuss this, plus preface it with a brief history of HDOS to bring those of you unfamiliar with the domain up to speed.

HDOS Origins

The origins of this system lie in the '70's, though certainly not as early as Gary Kildall's CP/M. To refresh your memory, CP/M was first written for Intel's 8008 MPU in '71 to use IBM's new 8-inch floppies. Rewritten for the 8080, Kildall's marketing company, Digital Research, finally turned the financial corner in '75 as the idea of an OS standard caught on. Anyway, Heath Company shipped its first digital computer, the venerable H-8, in the fall of 1977. As with so many of those earliest micros (the Apple II, Radio Shack TRS-80 Model I, and Commodore PET come to mind), external storage was on audio cassette. Though Heath offered an irascible paper tape punch/reader, too. But the poor reliability of both these medias sent the designers out looking for better alternatives and one of these was the new 5¼ inch floppy drives just coming onto the market. Shugart was the only state-side supplier and the '77 to early '78 period seems to have been a race between the major micro manufacturers to see who would release the first floppy subsystem. Apple beat out the likes of Commodore and Radio Shack (not to mention Heath), but the Shugarts were themselves not yet the ultimate in trustworthiness, either.

Kirk L. Thompson is the editor of The Staunch 8/89'ers, *a bi-monthly newsletter on 8-bit H/Z computers. The sample issue I have is twelve pages of good solid technical information for Heath/Zenith owners. It is published 6 times a year, and the rate is $12 for one year (plus $4 overseas), or $24 for two years (plus $8 overseas). Subscriptions always start and end with the calendar year. Make checks payable to Kirk L. Thompson. You can reach him at #6 West Branch Mobile Home Village, Route 1, West Branch, IA 52358 (319) 643-7136.*

So half a year after its release, a floppy-disk sub-system for the H-8 was coming off the drawing boards (using a more reliable drive from German maker, Siemens/Wangco) and this raised the specter of system management. The question was whether to write something with the H-8's unique architecture (an 8K ROM monitor at the bottom of memory containing the cassette I/O routines) in mind, or adopt the rising star, CP/M ver. 1.4. Heath chose upward compatibility and reliability (CP/M 1.4's reputation in the latter department was none too good and writing an OS is easy, right?) and contracted with an outside consulting firm, Wintek of Lafayette, IN. The man Wintek sent was J. Gordon Letwin.

It didn't take long for Heath to realize that J.G.L. was one super programmer. So they "pirated" him from Wintek. But the hurdles this young man were to face during development and maintenance of HDOS were the likes that none of the current generation of application developers is likely ever to face, thank goodness. The system was developed on a Digital Equipment UNIX-based PDP-series mini, cross-assembled for the 8080 MPU, and downloaded to the H-8 for testing. That first version (HDOS 1.0) was written by J.G.L. in six weeks! In addition to the OS, an assembler, line editor, debugger, a dialect of BASIC, and a set of utilities were written. True to Heath's tradition, the system was designed for the inexperienced and the documentation was voluminous and next to superb. And contrary to popular opinion, HDOS's heritage lies with the OS on the DEC Letwin developed the system on. A further bonus to Heath's clientele lay a few years further up in the stack when source code for the system was sold.

All things change, at Heath as well as elsewhere. J.G.L. was, in turn, "pirated" by a rising software firm named MicroSoft. And principal HDOS system enhancement was assumed by Gregg A. Chandler. By this time, Heath had also developed another hardware platform, the H-89, and been acquired by Zenith. Like the H-8, the '89 retained an 8K monitor at the bottom of memory, though the '8's seven-segment display and 16-key keypad were scrapped in favor of a full-screen CRT with keyboard. Indeed, the '89 was built **inside** the cabinet of Heath's most popular terminal of the time, the H-19.

True CP/M compatibility was also in the offing. For a time, Lifeboat Associates offered a modified ver. 1.4, ORiGinating above the ROM monitor, at 2280hex. Though early versions (such as my own '89) could only boot HDOS, when the F.C.C. mandated more stringent control of RFI in 1981, the redesigned '89A boasted memory remapping for ORG-0 CP/M 2.2. (I was able to retro-fit my own machine.) But the H-8 did not survive this shakeout and Heath discontinued production in '81, though it continued to sell them from stock for several years. However, it did offer a replacement CPU board for the machine that duplicated the '89's Z80 and memory remapping so it, too, could boot CP/M 2.2 when

that was released.

Besides the new platform, HDOS matured during these years, though not in the way CP/M did over the same period. This was because of the philosophical differences between the two. The system became even more device-independent with the separation of the floppy disk driver routines from the system core. (I recall that within a couple years after I bought my "Neanderthal," I received three versions of HDOS, 1.5, 1.6, and 2.0.) A good share of the motivation for this was the immanent release of soft-sector controllers for both the '8 and '89 in 1982. Heretofore, HDOS could only boot from single-density, ten-hard-sector, single-sided floppies. The addition of soft-sector format expanded capacity from about 90K to upwards of 760K per 5¼ floppy, depending on the number of sides and track density of the drives installed. This addition also meant that the eight-inch floppy subsystem Heath had supported for some years could now also produce HDOS system disks.

But this marked the last revision prepared directly by Heath, upgrading the system from ver. 1.6 to 2.0. A further update, explicitly to support a just-released hard disk system (the H-67) was promised. But Heath now had other irons in the fire, specifically, development of the H-100 dual-processor (8085/8088) system and its programmers were shunted away from further HDOS work.

HDOS 3.0 Origins

As I mentioned, Heath promised an update beyond HDOS 2.0. Indeed, one of the persistent questions at Heath/Zenith-oriented conferences (HUGCONs), sponsored by Heath Co., Zenith Data Systems, and the Heath/Zenith Users' Group (HUG), during the mid-80's was, "Where's HDOS 3.0?" Actually, development was let to a small group of independent contractors, organized by Bob Ellerton (then manager of HUG), lead by William G. Parrott III and David T. Carroll, and originally included Dale Wilson. As initially conceived, it was to boot from all three of Heath/Zenith's hardware platforms: the H-8 (with either the original 8080 or new Z80 MPU card installed), '89, and '100. Besides support for hard disks, the one significant wrinkle that would separate it from earlier versions of HDOS (or CP/M, for that matter) was that the system core would reside at the **bottom** of memory. This would require memory remapping on the '8 and '89, as is done when booting CP/M 2.2, to remove the monitor ROM from active memory. But it would also provide compatibility with most of the software written for HDOS 2.0 and earlier since program ORiGin (at 2280hex) **wouldn't** change. Unexpectedly (as I discovered after getting the system), compatibility extended to many third-party printer device drivers even though alterations to the PIC (position-independent code) structure (permitting dynamic positioning of driver modules in memory) were made.

Development ran into significant problems almost immediately, though. David Carroll managed to acquire the system source code on magnetic tape and uploaded it to 8-inch disks, providing it to Bill Parrott. After this, machine-readable copies of the HDOS 2.0 or earlier source seem to have disappeared. On the bright side, the command processor (SYSCMD.SYS, comparable to CP/M's CCP) wasn't based directly on HDOS 2.0's original, but on a dramatically improved version developed by Bill and Dave while employed by D.G. Electronic Development. But with increased use of early versions of MSDOS on the 16-bit side of the H-100, a version for the 8085 was abandoned, leaving CP/M-85 as the only 8-bit system for that machine. Further, after the team leaders had designed the system, the agreement with Heath collapsed, so the project lay dormant for a period. However, both Parrott and programmer Richard Musgrave (more on him below) belonged to the Mission (KS) Users' Group (MUG) and Musgrave "bugged" Parrott into continuing the project with his assistance. Musgrave wrote me recently that this experience converted him into a systems programmer! At any rate, development of 3.0 became very much a part-time, hobbyist affair. One of the few bright spots was release by Dean Gibson of UltiMeth of an advanced assembler for HDOS. (This, by the way, is still available from Quikdata, Inc., in Sheboygan, WI.) When eventually released, the source for HDOS 3.0 was written specifically for this assembler.

But despite difficulties, the upgrade eventually made it into the users' hands: in two forms and unexpectedly placed in the public domain! Release was announced by Bob Ellerton at HUGCON V in August, '86. It could either be downloaded from HUG's BBS as huge ARC'd files (requiring deARCing under MSDOS or CP/M and transfer to HDOS-compatible media!) or ordered in bootable form directly from team leader Bill Parrott as a seven-disk set for $25. This included the system, utilities, BASIC, assembler, and source for a wide range of device drivers. I took the latter route in December that year and apparently from its release until Bill left the team to pursue other interests in mid-'88, some 750 copies were sold. I have no idea how many downloaded the ARC files from HUG's BBS. The only drawback to the release was the skimpy documentation. It **presumed** knowledge of HDOS 2.0 and this created problems for those coming from a CP/M background.

HDOS 2.0 Today

Besides the release of version 3.0, 1986 marked something of a turning point for Heath's old, now-discontinued 8-bit systems and HDOS. For one thing, two newsletters specifically for the H-8 and '89 made their appearance. The first was Leonard Geisler's *SEBHC Journal*, appearing in August as a monthly. This was followed by my friend Hank Lotz with his quarterly, *The Staunch 8/89'er*, in November. For another, Hank had asked me to write a column for him and during the following year, I became increasingly concerned with the status of HDOS 2.0 and the rather evident decline in HDOS support from both Heath and third-party software vendors. So several months before Hank and I swapped places on the newsletter's "staff" in January, '88, I wrote to Heath, HUG, and Zenith Data Systems, encouraging them to release HDOS 2.0's source and object code into the public domain. (At this point, I frankly hadn't given any thought to that voluminous documentation I mentioned above!)

In September, '87, Heath president William Johnson replied politely that Heath would consider my proposal. And the matter rested there for some six months while I acclimated myself to meeting publishing deadlines (even soft ones). But late in April, I received a letter from Bob Ellerton (now manager of Consumer Publications for Heath) stating that the company **was** placing the system in the public domain! He also implied that Heath was searching its archives for further materials and anything discovered would be placed on HUG's BBS. Regrettably, that search was in vain, but all wasn't lost!

Let's go back down in the stack for a moment. Recall that I mentioned (apparently off-handedly) that Heath sold listings of the HDOS source. A sizable number of people took advantage of that. A few wrote and sold clones of the system that used less memory than the original, although they usually could only boot from Heath's hard-sector disks. Others enhanced the command processor (SYSCMD.SYS) to abbreviate the typing and improve functionality. But of greatest interest for us was the apparent labor of love that members of the largest local Heath-oriented users'

group (the Capitol Heath/Zenith Users' Group [CHUG], located in the nation's capitol) undertook: keying the entire source! The rest of us owe these unsung, but intrepid, hackers considerable credit for the vital task they performed. And even though Bob Ellerton expressed caution over the possibility of assembling that source with the standard HDOS assembler in his letter releasing it—he speculated that the UltiMeth assembler would be required since Letwin and his successors used DEC's RSTS—one of my readers recently wrote that he was able to assemble the system, using HDOS's standard assembler, from CHUG's source code with but a few minor changes.

So the system was now available. But describing HDOS to new users is a major problem, whether they have previous experience with CP/M or MSDOS or are complete novices. Though easy to use once you know it and having messages explicitly describing errors, HDOS requires more "expression" on the part of the user. (Hence the popularity of modifications abbreviating commands.) But the question of documentation was resolved fairly quickly. And its prompting was by one of my readers, Parks Watson. In response to his question, was the HDOS documentation also public domain, I forwarded his query to Jim Buszkiewicz, Managing Editor of *REMark* (the magazine published by HUG) and Jim replied in June of 1988 that it, indeed, was! Like the prior publication of the HDOS source, this, too, had some unexpected consequences, which I'll turn to momentarily.

But this release immediately prompted me to arrange for an easier distribution medium for the manual, particularly to those acquiring the '89 second-hand but without any operating system. I contracted with Daniel Jerome, a semi-retired technical writer, to keyboard and update the whole thing! And the latter was certainly overdue. Its last such was in 1980, but the system saw further development through the first few years of the decade, as I remarked above, mainly associated with the release of the soft-sector controller boards. Information about the latter never made it into the HDOS documentation. Neither did material on the "undocumented" features of the OS, particularly STAND-ALONE mode for single-drive systems.

But to summarize, through Heath Co.'s largess, the entire HDOS 2.0 system (source code, object code, and documentation) is in the public domain. And they can be had from me. But I could wish that other manufacturers would be so generous, such as Digital Research. But I gather that CP/M 2.2 is **still** a hot seller, particularly in industrial and military applications.

HDOS 3.0 Today

There is some surprising activity involving the "child" of HDOS as well. Richard Musgrave (of MIGHTY-SOFT, Kansas City, MO), the other half of Bill Parrott's HDOS 3.0 development team, has continued to enhance the system and is presently preparing ver. 3.10 for release. And the public domain status of the 2.0 documentation has prompted writer Dan Jerome to combine with Richard in preparing an extensive revision of it specifically for 3.02, the current release, merging it with hefty document that Richard prepared himself. This package will be available from Quikdata (Sheboygan, WI) and me in the second quarter of this year. The revised documentation will run to something like 600 pages when printed from disk.

And William Lindley (Lindley Systems, Woodbridge, VA), one of the remaining long-time vendors supporting the H-8 and H/Z-89 on both the HDOS and CP/M sides, has volunteered to prepare camera-ready material for a hardcopy version with Ventura Publisher on his laser printer-equipped PC system. This combined effort will repair the one major deficiency still present in version 3.0.

HDOS'S Longevity

HDOS's staying power is nothing short of miraculous, considering the limited manufactured hardware base it was written for. One of the reasons is certainly the persistence of its users to stay with not only a known, but a **simple**, operating system. The same, of course, holds for today's many CP/M users. (Some of my subscribers have mentioned that though they must use PC-clones at work, they prefer the accessibility of the 8-bit system, whether HDOS or CP/M, at home.) Another is undoubtedly Heath Co.'s exceptional and enlightened support, even though both hardware platforms have been out of production for half a decade.

But the main reason is, I think, Letwin's original design concepts for the system: user friendliness--no obscure error messages are displayed when there's a problem--and device independence. The latter is certainly the more important of the two and means that when you buy a new peripheral, you need **not** patch the system as is the case with CP/M. You just have to acquire or write a device driver for it. When the peripheral is used, the driver is automatically loaded into memory below the system, used according to its function, then that memory is **reclaimed** by the system for other things.

Given the changes in Microsoft's MSDOS, beginning with version 2.x, Letwin appears to be applying the experience he acquired developing HDOS to the 16-bit environment. And in that sense, the heritage of HDOS will survive well beyond the moment when the last H-89 is retired to the Smithsonian.

Acknowledgment: Although the sources which contributed to the discussion above were wide-ranging, I would particularly like to thank Richard Musgrave, co-developer of HDOS 3.0. He proofed the first draft and contributed significantly to my discussion of the origins of HDOS. •

MOVING?

Make certain that TCJ follows you to your new address. Send both old and new address along with your expiration number that appears on your mailing label to:

THE COMPUTER JOURNAL
190 Sullivan Crossroad
Columbia Falls, MT 59912

If you move and don't notify us, TCJ is not responsible for copies you miss. Please allow six weeks notice. Thanks.

The Z-System Corner

by Jay Sage

This time my column is going to be quite short. In response to my requests, a number of authors have submitted some very interesting articles, but there has not been enough space to print them. I want to make sure that those articles are not delayed further. One of them is on the superb LSH history shell by Rob Friefeld, who has contributed quite a number of excellent Z-System programs (SALIAS, VCOMP, and BCOMP, to name a few). You should not miss that article.

After working first with the original Z-System history shell (HSH by Michael Rubenstein) and then with EASE by Paul Pomerleau, it occurred to me that it would be even nicer to have a full-screen history shell. What I envisioned was bringing the full resources of a wordprocessor to bear on the command transcript, so that commands could be easily viewed, modified, reordered, and regrouped. If the history file were a standard ASCII file, then one could massage the file with a standard editor or even prepare 'history' scripts in advance for special purposes.

After seeing the splendid full-screen work Rob Friefeld had done in his SALIAS (Screen ALIAS editor), I asked him if he would take on the task of writing such a history shell. He did, and he has done a splendid job. I would, therefore, like to publicly take credit for that all-important management skill of asking the right person to do a job!

Software Update Service

While Echelon was still in business marketing the Z-System, they offered a very nice product called SUS or Software Update Service. People who have modems and a nearby Z-Node or RCP/M system generally do not have much trouble picking up the latest releases of public-domain Z-System and general CP/M software. However, for those who do not have modems or for whom the nearest Z-Node is an expensive long-distance call, obtaining a full set of Z-System tools or keeping up with new releases is much more difficult. The Echelon SUS was designed to solve that problem by making the material available on diskette by mail. It was a disk subscription service, and roughly every month subscribers would get a diskette full of public-domain software.

I am happy to announce that SUS is coming back, thanks to the urging and energy of Chris McEwen, sysop of the Socrates Z-Node (#32), in Plainfield, NJ. Chris and Bill Tishey, together with Sage Microsystems East, will be offering an even more extensive service than Echelon's. Bill Tishey, as most of you know, has for some time been maintaining a complete catalog of Z-System programs (ZFILESnn.LST) and a compendium of HLP files covering all of them. At frequent intervals, Bill releases an update LBR with all the new help files. Now, in addition to that service, Bill will be putting together diskettes with the software as well as the documentation.

This means that you will be able to purchase diskettes with the complete set of Z-System programs and/or subscribe to a monthly update service. Bill and Chris will be handling most of the diskette production; SME will handle the orders and bookkeeping and will produce diskettes in the few formats that Chris and Bill cannot handle (8" IBM SSSD, NorthStar hard-sector, and Amstrad 3").

We have not yet worked out all the pricing details for all the options, but by the time you are reading this column, we will have flyers available with all the information. Just drop me a letter or postcard, or leave a message for me in any of the ways indicated in the sidebar to this column, and I will get a flyer to you. To give you some idea of what we are talking about, a 6-month SUS subscription to a US address will probably be $48 ($8 per disk) and a year's subscription $72 ($6 per diskette). As you can see, we are trying to keep the price very low. We really want all of you to be able to get and use all these wonderful programs.

Fully Customizing NZCOM

My technical topic for this time will be about designing fully customized NZCOM Z-Systems. I have always been satisfied with the systems that can be produced so easily using the MKZCM (MaKe nZCoM) menu-driven utility, and so I never really delved into this area very much. About a week or so ago, however, Dave Goodman brought the problem to me. He has a NorthStar Horizon with an add-on hard disk, and the operating system has a ROM stuck somewhere in the middle of the address space. That left some disjoint blocks of free memory, and Dave really wanted to make use of all the space. I told him my standard answer to that problem.

In section 5 (especially subsection 5.2.3) of the NZCOM manual, I point out that the NZCOM system is defined by a descriptor file and that this file (with type ZCM) is a pure ASCII file that can be edited with one's favorite text editor. The manual recommends that everyone make certain changes so that the descriptor will properly reflect the user's hardware environment, such as the disk drives available and the characteristics of the system's printer and terminal.

I did not actually come out and say it explicitly, but there is an implication that other values in the ZCM file can also be changed. The truth is, I believe, that I avoided this subject in part because I

Jay Sage has been an avid ZCPR proponent since the very first version appeared. He is best known as the author of the latest versions 3.3 and 3.4 of the ZCPR3 command processor and for his ARUNZ alias processor and ZFILER point-and-shoot shell.

When Echelon announced its plan to set up a network of remote access computer systems to support ZCPR3, Jay volunteered immediately. He has been running Z-Node #3 for more than five years and can be reached there electronically at 617-965-7259 (MABOS on PC-Pursuit, pw=DDT). He can also be reached by voice at 617-965-3552 (between 11pm and midnight is a good time to find him at home) or by mail at 1435 Centre St., Newton, MA 02159. Jay is now also the Z-System sysop for the GEnie CP/M Roundtable and can be contacted as JAY.SAGE via GEnie mail or chatted with live at the Wednesday real-time conferences (10pm Eastern time).

In real life, Jay is a physicist at MIT, where he tries to invent devices and circuits that use analog *computation to solve problems in signal, image, and information processing. His recent interests include artificial neural networks and superconducting electronics. He can be reached at work via Internet as SAGE @LL.LL.MIT.EDU.*

```
E606 CBIOS      0080 ENVTYP     E3F4 EXPATH     0005 EXPATHS    D300 RCP
0014 RCPS       0000 IOP        0000 IOPS       DD00 FCP        0005 FCPS
DF80 Z3NDIR     0023 Z3NDIRS    E400 Z3CL       00CB Z3CLS      E280 Z3ENV
0002 Z3ENVS     E200 SHSTK      0004 SHSTKS     0020 SHSIZE     E380 Z3MSG
E3D0 EXTFCB     E4D0 EXTSTK     0000 QUIET      E3FF Z3WHL      0004 SPEED
0010 MAXDRV     001F MAXUSR     0001 DUOK       0000 CRT        0000 PRT
0050 COLS       0018 ROWS       0016 LINS       FFFF DRVEC      0000 SPAR1
0050 PCOL       0042 PROW       003A PLIN       0001 FORM       0000 SPAR2
0000 SPAR3      0000 SPAR4      0000 SPAR5      BB00 CCP        0010 CCPS
C300 DOS        001C DOSS       D100 BIO        0000 PUBDRV     0000 PUBUSR
```

Figure 1: The ZCM descriptor file for the normal NZCOM system I use on my Televideo 803H computer.

```
E606 CBIOS      0080 ENVTYP     E3F4 EXPATH     0005 EXPATHS    D700 RCP
0014 RCPS       0000 IOP        0000 IOPS       D480 FCP        0005 FCPS
D200 Z3NDIR     0023 Z3NDIRS    E400 Z3CL       00FB Z3CLS      E180 Z3ENV
0002 Z3ENVS     E100 SHSTK      0004 SHSTKS     0020 SHSIZE     E280 Z3MSG
E2D0 EXTFCB     E300 EXTSTK     0000 QUIET      E2FF Z3WHL      0004 SPEED
0010 MAXDRV     001F MAXUSR     0001 DUOK       0000 CRT        0000 PRT
0050 COLS       0018 ROWS       0016 LINS       000F DRVEC      0000 SPAR1
0050 PCOL       0042 PROW       003A PLIN       0001 FORM       0000 SPAR2
0000 SPAR3      0000 SPAR4      0000 SPAR5      BA00 CCP        0010 CCPS
C200 DOS        001C DOSS       D000 BIO        0000 PUBDRV     0000 PUBUSR
```

Figure 2: A radically reconfigured NZCOM system produced by manually editing the ZCM file.

was not entirely sure which values could and which values could not be changed. My suggestion to Dave Goodman was that he experiment with designing a custom memory map for his system, edit the values into the ZCM file, and see what happened when he tried to load it.

Dave's report back to me, now confirmed by my own experiments on my Televideo 803H, indicated that ALL values can be changed. The only requirement is that the values provide a memory map with no modules overlapping. When you use MKZCM to design the system, it takes over the responsibility for generating a valid memory map; if you do the design yourself, you better be careful.

A Helpful Utility

This suggests a very nice utility program that some thoughtful soul could contribute to the community. This utility (let's call it ZMAP) might do a number of helpful things. First, it could display, perhaps in some graphical or semi-graphical way, the memory map of a Z-System, the one actually running or one specified in the form of a ZCM or ENV file (and maybe even the Z3PLUS descriptor file of type Z3P). Present utilities, such as SHOW (ZSHOW) and Z3LOC, list the module addresses in a fixed order, not in order of increasing memory address. Thus they are not very helpful in determining if there are gaps or overlaps in the map. Ideally, ZMAP would flag any such defects or potential defects in the map so that they could be corrected before they cause harm.

The final item on my wish list—and this might better be implemented in a second, independent program (ZDESIGN perhaps)— would be a general Z-System designer, along the lines of MKZCM but without its restrictions. One would be able to specify the order of all the modules in memory and their sizes. Given the highest memory address available, the program would then figure out and display the memory map. One should be able easily to alter the order of the modules, and one should be able to override specific addresses to create gaps if necessary (but not to force overlaps). Once the desired system has been designed, the program should write out a ZCM or ENV file for it. Such a program is a good candidate for implementation with a high level language such as BDS Z or Turbo Pascal. And it sure would have helped me with the experiments that I am about to describe (several mistakes resulted in crashes).

My Experiments

Figure 1 shows a printout of the standard NZCOM.ZCM file on my Televideo 803H. It has already been customized in several ways using MKZCM. First, it allocates a 4-record VBIOS. I use a version that fixes the 803's faux pas of clobbering the index registers during BIOS calls and implements a check of the Z-System drive vector for BIOS disk-select calls as described in a previous column. It also has room for a 20-record RCP, which allows me to use a full-featured RCP with Carson Wilson and Rob Friefeld's resident history shell, CLED (see RCPZRL11.LBR on Z-Nodes).

I decided to be cautious, especially after one of my new system designs caused the system to hang, and I made a series of systems, each different from the previous one in a relatively small way. I am not going to show you all the steps along the way but will go right to the most radically different version. See Figure 2. If you look carefully, I think you will find that only the command line buffer (Z3CL) is still in the same place as it was in the original system (but it is bigger now).

Perhaps you are wondering why I didn't make the most dramatic demonstration possible by changing absolutely every address (and perhaps size, too). Well, there was an extra constraint that I was exploring with this system. I am running ZDDOS, and I have specified that the clock driver be loaded into the so-called user buffer. I have even applied the NZCOM patch (NZCOMPAT.HEX) that comes with the ZSDOS/ZDDOS package so that when new system configurations are loaded, the clock driver will be reconnected to the DOS automatically without the need for running LDTIM again.

If you know a lot about Z-System, you will know that there is no such thing as a user buffer! The user buffer is a special creature of NZCOM; it is not defined in the Z-System environment descriptor (or—look closely—in the ZCM file). How, then, does one determine where this special gap in the memory map of an NZCOM system is located? That is exactly what I wondered myself. I could have called ZDOS authors Cam Cotrill or Hal Bower and asked them how they infer its location, but I decided to experiment instead. What I found after various trials and errors was that the NZCOM patch seemed to be happy and able to find the LDTIM clock module so long as the command line buffer stayed in the same place. Apparently, the assumption is made that the user buffer is the memory from 100H above the start of the command line buffer up to the real CBIOS (E400 to E5FF in my case).

I did not perform exhaustive tests of this hypothesis. Let us just say that it is not terribly prudent to try to make use of a 'user buffer' with a fully customized system. It would be wiser to design the system with a gap below the CBIOS for the clock driver and to create a version of LDTIM with an explicit load address. The NZCOMPAT patch should be omitted from NZCOM if such custom systems are going to be used.

A Few Bugs

There were a few bugs in NZCOM that surfaced during this testing that suggest that NZCOM.COM was not quite designed to work rigorously and to handle the most general system loading situations. Sometimes I found that NDR modules became empty, and the command search path was rarely preserved with these systems. Code-containing modules, such as the FCP, RCP, DOS, and so on, cannot be moved from one address to another. If their starting address changes, the code must be reloaded fresh from the ZRL file. On the other hand, modules that contain data, such as the NDR, shell stack, path, message buffer, and so on, can and should be moved to any new address, so long as there is room for the old contents in the new home. NZCOM sometimes failed to do this. Maybe now that I have uncovered these small problems, I can pass the information on to Joe Wright, and he can fix up the code to handle these situations. •

Graphics Programming With C On the IBM PC

by Clem Pepper

As ancient civilizations rose and fell the newest built on the ruins of their predecessors. By digging down through the layers much has been learned of the progress made through the ages. In many respects an exploration of the IBM PC bears a strong resemblance to an archaeological dig.

My latest in this series dealt with screen character attributes. This in the context of the TEXT mode of operation. When we screw up our courage and venture into the GRAPHICS mode we find ourselves in an entirely new world. It is as though we have two more or less unrelated computers sharing a single environment. Schizophrenic just may be the keyword in defining the relationship of the two modes. Dr. Jekyll and Mr. Hyde, if you will. Our computer can be in the text mode or the graphic, but not in both at the same time.

Video Monitors and Adapters

A monitor screen of some kind, CRT or liquid crystal, monochrome or in color, allows us to view the computer's output. IBM and compatible computers require a video adapter to translate computer instructions into a format capable of controlling the monitor. The adapter is a printed circuit card containing the video control logic. The card also contains the video memory.

In the beginning there was the Monochrome Adapter, the MDA. The MDA is limited to text only; no graphics. The Hercules Graphic Adapter (HGA) displays graphics in monochrome only with a resolution of 720 by 348 pixels.

To meet the demand for color IBM came up with the Color Graphics Adapter (CGA). In its black and white (2-color) mode this adapter has a resolution of 640 by 200 pixels. Its color resolution is 320 by 200 pixels. It is subject to severe limitations on available colors. In its graphic mode, as distinct from text, two color palettes are available. (Turbo C has somehow contrived to provided four palettes.) Only one can be in use at any given time. A palette provides only three colors. The screen background color

MS DOS CGA Graphics Display Options:

Two possible resolutions: 300 x 200 pixels, 640 x 200. 640 x 200 in monochrome only.

In 320 x 200 each pixel may be one of four colors. Can be one of 16 background colors or one of three other colors selectable from one of two available color palettes.

Palette One:	Palette Two:
color 1 - RED	color 1 - CYAN
color 2 - GREEN	color 2 - MAGNETA
color 3 - BROWN	color 3 - WHITE

Available background colors:

0 - BLACK	1 - BLUE	2 - GREEN	3 - CYAN
4 - RED	5 - MAGNETA	6 - BROWN	7 - LIGHTGRAY
8 - DARKGRAY	9 - LIGHTBLUE	10 - LIGHTGREEN	11 - LIGHTCYAN
12 - LIGHTRED	13 - LIGHTMAGNETA	14 - YELLOW	15 - WHITE

Turbo C Version 2.0 graphics library CGA palettes for use with setcolor()

Palette Number	Constant assigned to color number (pixel value) 1	2	3
0	LIGHTGREEN	LIGHTRED	YELLOW
1	LIGHTCYAN	LTMAGNETA	WHITE
2	GREEN	RED	BROWN
3	CYAN	MAGNETA	LIGHTGRAY

In use pass the color number or name. In name, prefix the color with CGA_. Example: in CGAC1 mode, setcolor(3) or setcolor(CGA_WHITE) selects the color white.

Background colors and color numbers are as shown with MS DOS above.

Colors must be in upper case when used with setcolor().

Table 1. Palette and color options available in the CGA graphics mode.

Mode Constant	Interpretation	Adapter
CGAC0	320 x 200 palette 0, 1-page	CGA
CGAC1	320 x 200 palette 1, 1-page	CGA
CGAC2	320 x 200 palette 2, 1-page	CGA
CGAC3	320 x 200 palette 3, 1-page	CGA
CGAHI	640 x 200 two-color, 1-page	CGA
EGALO	640 x 200 16-color, 4-pages	EGA
EGAHI	640 x 350 16-color, 2-pages	EGA
EGA64LO	640 x 200 16-color, 1-page	EGA
EGA64HI	640 x 350 4-color, 1-page	EGA
EGAMONOHI	640 x 350 64K on card 1-page 256K video memory, 4-pages	EGA
HERCMONOHI	720 x 348 2-pages	HERCULES
VGALO	640 x 200 16-color, 4-pages	VGA
VGAMED	640 x 350 16-color, 2-pages	VGA
VGAHI	640 x 480 16-color, 1-page	VGA

Table 2. Turbo C Version 2.0 graphics library video modes.

Turbo C Version 2.0 EGA/VGA Color Table

Name	Value
EGA_BLACK	0
EGA_BLUE	1
EGA_GREEN	2
EGA_CYAN	3
EGA_RED	4
EGA_MAGNETA	5
EGA_LIGHTGRAY	7
EGA_BROWN	20
EGA_DARKGRAY	56
EGA_LIGHTBLUE	57
EGA_LIGHTGREEN	58
EGA_LIGHTCYAN	59
EGA_LIGHTRED	60
EGA_LIGHTMAGNETA	61
EGA_YELLOW	62
EGA_WHITE	63

Table 3. Turbo C Version 2.0 graphics library color table for EGA/VGA.

```
closegraph ..... shuts down the graphics system
detectgraph ..... checks the hardware and determines which
                  graphics driver to use; recommends a mode
graphdefaults ... resets all graphics system variable to
                  their default settings
_graphfreemem ... deallocates graphics memory; hook for
                  defining your own routine
_graphgetmem .... allocates graphics memory; hook for
                  defining your own routine
getgraphmode .... returns the current graphics mode
getmoderange .... returns lowest and highest valid modes for
                  specified driver
initgraph ..... initializes the graphics system and puts
                  the hardware into graphics mode
installuserdriver installs a vendor-added device driver to
                  the BGI device driver table
installuser font  loads a stroked font not known to the
                  graphics routines
registerbgidriver registers a linked-in or user-loaded driver
                  file for inclusion at link time
restorecrtmode .. restores the original (pre-initgraph)
                  screen mode
setgraphbufsize . specifies size of the internal graphics
                  buffer
setgraphmode .... selects the specified graphics mode, clears
                  the screen, and restores all defaults
```

Table 4. A summary of the graphic library functions for graphics system control functions

Drawing functions:

```
arc ..... draws a circular arc
circle ..... draws a circle
drawpoly ..... draws the outline of a polygon
ellipse ..... draws an elliptical arc
getarccoords .... returns the coordinates of the last call to
                  arc or ellipse
getaspectratio .. returns aspect ratio of the current
                  graphics mode
getlinesettings.. returns the current, line pattern, and line
                  thickness
line ..... draws a line from (x0,y0) to (x1,y1)
linerel ..... draws a line to a point some relative
                  distance from the current position (CP)
lineto ..... draws a line from the current position (CP)
                  to (x,y)
moveto ..... moves the current position (CP) to (x,y)
moverel ..... moves the current position (CP) a relative
                  distance
rectangle ..... draws a rectangle
setaspectratio .. changes the default aspect ratio-correction
                  factor
setlinestyle .... sets the current line width and style
```

Filling functions:

```
bar ..... draws and fills a bar
bar3d ..... draws and fills a 3-D bar
fillellipse ..... draws and fills an ellipse
fillpoly ..... draws and fills a polygon
floodfill ..... flood-fills a bounded region
getfillpattern .. returns the user defined fill pattern
getfillsettings . returns information on the current fill
                  pattern and color
pieslice ..... draws and fills a pie slice
sector ..... draws and fills an elliptical pie slice
setfillpattern .. sets the fill pattern and fill color
```

Table 5. A summary of the graphics library functions for drawing and filling.

Screen manipulation functions:

```
cleardevice ..... clears the screen (active page)
setactivepage ... sets the active page for graphics output
setvisualpage ... sets the visual graphics page number
```

Viewport manipulation functions:

```
clearviewpoint .. clears the current viewport
getviewsettings . returns information about the current
                  viewport
setviewport ..... sets the current viewport for graphics
                  output
```

Image manipulation functions:

```
getimage ..... saves a bit image of the specified region
                  to memory
imagesize ..... returns the number of bytes required to
                  store a rectangular region of the screen
putimage ..... puts a previously saved bit image on the
                  screen
```

Pixel manipulation functions:

```
getpixel ..... gets the pixel color at (x,y)
putpixel ..... plots a pixel at (x,y)
```

Table 6. A summary of the graphic library functions for image manipulation.

Graphics mode text output functions:

```
gettextsettings . returns the current text font, direction,
                  size, and justification
outtext ..... sends a string to the screen at the current
                  position(CP)
outtextxy ..... sends a string to the screen at the
                  specified position
registerbgifont . registers a linked in or user-loaded font
settextjustify .. sets text justification values used by
                  outtext and outtextxy
setextstyle ..... sets the current text font, style and
                  character magnification factor
setusercharsize . setswidth and height ratios for stroked
                  fonts
textheight ..... returns the height of a string in pixels
textwidth ..... returns the width of a string in pixels
```

Table 7. A summary of the graphic library functions for graphics mode text output.

is independent of the palette colors. CGA color options are described in Table 1.

The CGA was followed by the EGA for higher resolution and a wider range of colors. EGA resolution is 640 by 350 pixels with up to sixteen colors. The latest additions to the list are the MCGA and VGA adapters. These were initially developed by IBM for their PS/2 systems. These two adapters are analog in contrast to the digital ones used by the preceding cards. An analog monitor is required with these. Tables 2 and 3 describe Turbo C version 2.0 color options for the various adapters in use.

All the video adapters are "memory mapped." Each picture element (pixel) on the screen corresponds to one or more bits of the video memory on the adapter card. Although the memory is on the adapter card it is addressable in the same manner as memory elsewhere in our computer. This is good in that it allows our programs to produce screen displays by sending instructions directly to the adapter RAM.

Tables 4 to 6 include function definitions for pixel color control. Understanding the color functions requires some awareness of how colors are produced on the graphics screen. This screen consists of an array of pixels each producing a single colored dot via an index into a color table (the palette). The palette entry defines the exact color for that pixel. Use of the palette introduces a number of restrictions in that only a subset of colors can be made available at a given time. For the CGA in its low resolution (350 by 200 pixels) we can choose four colors from a total of 16 supported by the hardware; for the EGA with its higher resolution the palette provides for 16 out of the 64 available.

In graphic modes the screen pixels are organized in a matrix having an x-y coordinate scheme. The number of pixels, the colors available and the screen aspect ratio (ratio of horizontal to vertical pixels) varies with the adapter in use. Depending on the compiler and/or graphics library we are using the upper left corner of the screen may or may not be 0,0. In this respect the Turbo C graphics use the upper left corner as origin. This is true of the screen as a whole and viewports, to be described later. The library provided with the book *Graphics Programming In C* by Roger T. Stevens employs offsets in some of its plotting routines.

The ROM BIOS

One remaining component is the ROM BIOS, discussed in the previous article. The BIOS provides very limited graphics support. And no support whatever for the Hercules adapter. Use of the interrupt 10H routines do ensure portability, but with a severe penalty in performance. For fast screen drawing and animation direct writing to the video RAM is often preferable to portability.

Getting Started

As with so many of life's experiences there is an initial barrier arising from the unknown that may cause some of us to put off that first step into graphics programming. With graphics we are in new territory with respect to screen factors, memory use, and simply knowing how to utilize the computer power available to us.

An early obstacle to my own experience was how to keep track of objects on the screen. There are but a mere 70,000 pixels available with the CGA in its four color mode. A tracking scheme of some sort became my first priority. The solution was to draw up a screen map with a point for each pixel. Figure 1 shows the upper left portion of the map. Through use of the map we can speed up our designs considerably. And ease the task of placing our graphic objects on the screen.

In use I lay tracing paper over the grid and sketch out the object to be created. Figure 2 illustrates the technique with the design of a fighter aircraft. Listing 3 translates the design into a graphics display. A full scale grid can be made by photocopying the figure repeatedly and taping the segments to obtain a complete screen layout.

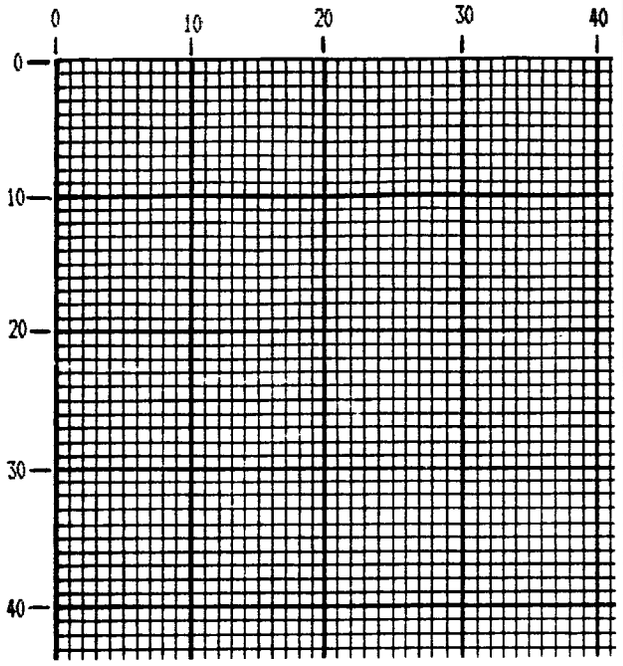
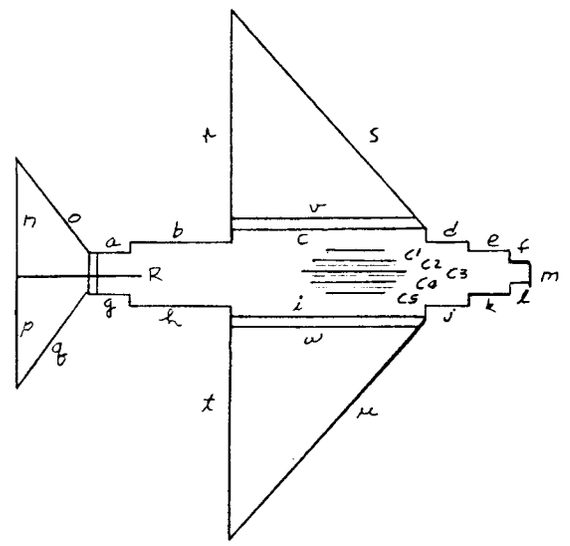


Figure 1. A portion of the screen grid having a one pixel resolution.

TURBO C Graphics

It is unfortunate that the standardization existent with the text mode is not true for graphics. So I am compelled to use what I have available in a manner that hopefully can be related to other libraries in use by readers. From what I have seen the graphic features are themselves close to universal, but function names and how to apply them may differ. With this as a basis it should not be too difficult to apply the examples given here to compilers having similar graphic capabilities.

Version 1.0 did not provide for graphics in its library functions.



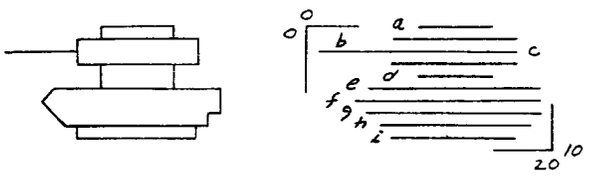
Line	From	To
Fuselage:		
a	8,23	11,23
ag	8,23	8,27
b	11,22	21,22
c	21,21	40,21
d	40,22	44,22
e	44,23	48,23
f	48,24	50,24
g	8,27	11,27
h	11,28	21,28
i	21,29	40,29
j	40,28	44,28
k	44,27	48,27
l	48,26	50,26
m	50,24	50,26
Cockpit:		
c1	30,23	36,23
c2	29,24	37,24
c3	28,25	38,25
c4	29,26	37,26
c5	28,27	36,27

Line	From	To
Horizontal Stabilzer:		
np	0,14	0,36
q	0,36	7,27
qo	7,27	7,23
o	7,23	0,14
Vertical Stabilizer		
R	0,25	12,25
Wings:		
r	21,21	21,0
s	21,0	39,20
v	21,20	39,20
t	21,29	21,50
u	21,50	40,29
w	21,30	39,30

Figure 2. Construction data for the aircraft of Listing 3.

Version 1.5 does. These were modified considerably for version 2.0. The book by Stevens was written using version 1.0. and checked with version 1.5 according to the author. The demo on the disk provided with the book and programs I have written based on its routines run on my version 2.0. Because of differences in coordinate schemes and the powerful graphics routines now available in version 2.0 I am presently using only the TURBO graphics. If you purchase the Stevens book be sure to obtain the disk also as it contains library functions for the CGA, EGA, VGA and Hercules adapters. Overall the book is an excellent source, but it does require some effort to use as Stevens was strong in providing good routines but a bit short at times in describing their application. I found the source code for the demo program, which is included in the book, a valuable aid in clarifying several routines with which I had problems.

The book I make extensive use of with version 2.0 is *The Waite Group's Turbo C Bible* by Nabajyoti Barkakati. Where the Borland Reference Guide lists the library functions alphabetically, this book is organized by functions. Four chapters are devoted to graphics: 17, 18, 19, and 20. Chapter 17 covers graphics modes, coordinates, and attributes. 18 describes routines for drawing and



planning sketch

viewport positioning

Line	Width	From	To	Color
a	1	9,0	15,0	red
b	1	1,2	7,2	brown
c	3	7,2	17,2	red
d	1	9,4	15,4	red
e	1	5,5	19,5	green
f	1	4,6	19,6	green
g	1	5,7	19,7	green
h	1	7,8	18,8	green
i	1	7,9	17,9	green

Figure 3. Construction detail and coordinates for the tank object of listing 5. Sketch at the left translates into parallel lines for screen display.

animation. 19 shows how to combine graphics and text. 20 delves deeper into text routines.

A Brief Tour of the Turbo C Graphics Library

There is but a single graphics library. Functions are prototyped in the header file GRAPHICS.H which must always be included. In contrast to the text library the graphics library is not read routinely by the compiler. Instead it must be explicitly requested. To use the graphics functions with the Integrated Environment (TC.EXE) toggle Options/Linker/Graphics to On. With the command line (TCC.EXE) follow the program name with GRAPHICS.LIB. That is, TCC MYPROG GRAPHICS.LIB. The graphic routines all use far pointers, because of this graphics are not supported in the TINY memory model.

What we look for in the library are functions to ease our task of creating graphic objects for screen drawing and animation. Tables 4 to 7 summarize the functions provided in the Turbo C Version 2.0 graphics library.

The tables are organized by the nature of the functions. Table 4 lists graphics control functions. These include functions to initiate the graphics system, provide for screen and hardware requirements while in the graphics mode and return to the text mode. This table, and those following, are from the Turbo C User's Guide.

Table 5 is a summary of routines for drawing and filling. These include arcs, bars, circles, ellipses, lines, rectangles and the like. Functions for cursor positioning, moving, and connecting points are also given.

Table 6 includes functions for image manipulation on the screen. These include clearing the screen, setting up active pages, creating and clearing viewports, and saving and restoring images from memory.

Table 7 lists functions useful for combining text with graphics. The library includes four "stroked" font files: GOTH.CHR, LITT.CHR, SANS.CHR, and TRIP.CHR. A default 8 by 8 pixel font is built into the graphics system. The .CHR fonts are linked in after first converting them to .OBJ files with the BGIOBJ utility. Both horizontal and vertical text are supported. The default is horizontal. If we wish to create special fonts they will have to be called out by means of our own devising, such as elements of an array.

Example Programs

Four example programs are provided. The purpose of these is to identify some essential procedures for working in the graphics mode as well as to illustrate the use of several graphics features for screen drawing, filling, and animation.

Listing 1 (heading.c) is a heading common to all the programs. Note that main() is included. The graphic functions shown enable the graphic driver appropriate to your adapter (graphdriver), select the color palette (graphmode), and enable testing for errors arising from misuse of a graphics function in your program (errorcode). For convenience the call to set the background color to blue is also included. The background can be any color permitted for the adapter in use.

initgraph(&graphdriver, &graphmode, "your C location"); initializes the graphic system. My C compiler files are in directory E:\\BTC20. (The double \\ is required whenever a \ is employed in a C string.) As shown the system is CGA. If your system is not CGA this should be changed. Graphdriver may also be simply set equal to DETECT. The graphics system will then detect the kind of hardware in your system and properly initialize it. This feature is valuable for programs likely to be run on more than one adapter

```
These functions define the rectangle size, border line style
and
border color:

 rectangle(int left, int top, int right, int bottom);
  setlinestyle(int linestyle, unsigned upattern, int
thickness);
   setcolor(int color);

        These functions define the interior fill style
        and fill color

        setfillstyle(int pattern, int color);
         floodfill(int x, int y, int border);
```

Line Pattern Availability:

Style	Value	Description
SOLID_LINE	0	Solid line
DOTTED_LINE	1	Dotted line
CENTER_LINE	2	Centered line (alternating dash, dots)
DASHED_LINE	3	Dashed line
USERBIT_LINE	4	User defines

Line Width Availability:

Width	Value	Description
NORM_WIDTH	1	Normal line, 1 pixel wide
THICK_WIDTH	3	Thick line, 3 pixels wide

Fill Style Availability:

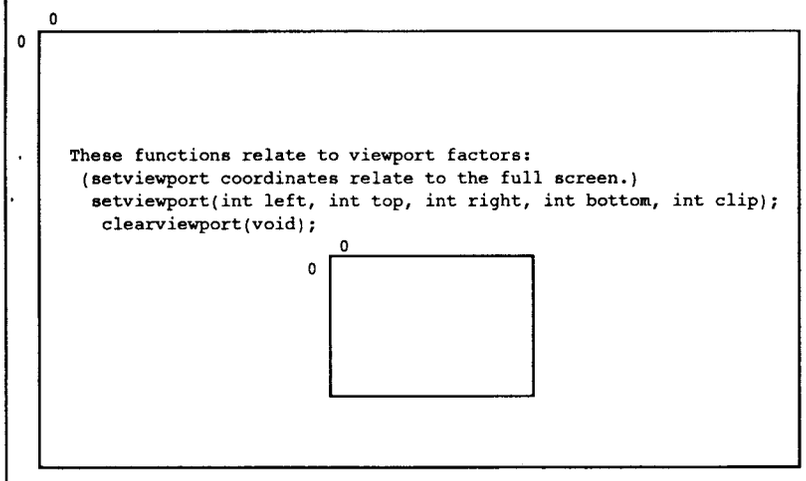
Style	Value	Description
EMPTY_FILL	0	fill with background color
SOLID_FILL	1	solid fill
LINE_FILL	2	fill with ----
LTSLASH_FILL	3	fill with ///
SLASH_FILL	4	fill with ///, thick lines
BKSLASH_FILL	5	fill with \\\, thick lines
LTBKSLASH_FILL	6	fill with \\\
HATCH_FILL	7	light hatch fill
XHATCH_FILL	8	heavy cross-hatch fill
INTERLEAVE_FILL	9	interleaving line fill
WIDE_DOT_FILL	10	widely spaced dot fill
CLOSE_DOT_FILL	11	closely spaced dot fill
USER_FILL	12	user defined fill pattern

Figure 4. Line style availablity and coloring in Turbo C Version 2.0 graphics.

Screen and viewport interior coordinates are both referenced from their upper left corners as seen in the sketch.

These functions relate to screen system factors:

```
initgraph(int far *graphdriver, int far *graphmode,
          char far *pathtodriver);
setaspectratio(int xasp, int yasp);
setbkcolor(int color);
cleardevice(void);
closegraph(void);
```



Multiple viewports with differing objects can be maintained on the screen.

Figure 5. Screen and viewport functions in Turbo C Version 2.0 graphics.

```
/* HEADING.C
** Include this listing with each of the example programs.
** Note that this listing includes main() {
*/

#include <stdio.h>
#include <graphics.h>

/* == Begin program == */
main()
{
int graphdriver = CGA;       /* graphics driver       */
int graphmode = 1;           /* specify 0, 1, 2, or 3 */
  /* ** graphmode cannot be changed later in a program ** */
int errorcode;               /* graphics error code   */

initgraph(&graphdriver, &graphmode, ''e:\\btc20'');
  /* ** replace ''e:\\btc20'' with your directory location **
*/
errorcode = (graphresult());     /* get result code */

  /* ** graphics error function routine call ** */
   if (errorcode != grOk)    /* always check for error */
  {
    printf(''Graphics error:
%s\n'',grapherrormsg(errorcode));
    exit(1);
   }

  /* ** call to set background color ** */
  setbkcolor(BLUE);
```

Listing 1. Heading to be used with the example programs. Change the graphmode value to match the program in use.

type.

The integer graphmode selects the palette to be used. This cannot be changed later in the program. The first example program, Listing 2, (rect.c) uses the rectangle drawing function to illustrate color selection and control for the colors available in a given palette. The listing shows palette 1; to view the other colors available change the palette number in your source code and recompile the program.

The program draws and fills four rectangles with distinctive border and fill colors. The code for the first rectangle is:

```
/* ** 1st rectangle(20,20,300,50) ** */
 setlinestyle(0,0,3);
  setcolor(1);          /* color 1 border */
   rectangle(20,20,300,50);

/* ** floodfill rectangle ** */
 setfillstyle(SOLID_FILL,2);  /* color 2 fill */
  floodfill(24,24,1);
```

The rectangle coordinates are screen pixels 20,20 for the upper left corner and 300,50 for the lower right corner. The line thickness is 3 pixels. The border color is value 1 of whichever palette is selected. Line patterns and thickness options are shown in Figure 4.

It is essential when using floodfill that there be no breaks anywhere in the figure's perimeter. This applies to any kind of shape you fill. A gap of but a single pixel will allow the color to "leak out" over the entire screen background. A solid fill is given in the program; other fill options are shown in Figure 4. The leading two numbers in floodfill(24,24,1); are pixel x,y coordinates at which the filling is to begin. These must be inside the border, anywhere you choose so long as it does not lie on the border. The third value is the color number for the border. The border can be any color, including the that for the fill.

The purpose of getch() in the programs is to hold the graphics screen for viewing until a key is pressed. Any graphics program we write must employ logic of some kind (getch() is impractical in a screen action game) to return to the text mode.

Listing 3 illustrates drawing a figure of your own design within a viewport. The object is the aircraft of Figure 2. It pays to lay out the line coordinates as shown prior to writing code. This holds even for simpler objects such as the tank of Figure 3.

A viewport is to graphics what a window is to text. We can show a border around the viewport but do not have to. This example does not.

Note the use of three drawing functions for positioning the aircraft: moveto(x,y), lineto(x,y), and linerel(x,y). The two end points are defined in move(x,y) and lineto(x,y); linerel(x,y) is the next endpoint relative to the coordinates given in the previous lineto(x,y).

Figure 5 describes the screen coordinates and functions for creating and using viewports. Note that a viewport has its own origin, 0,0, at its upper left corner. The function setviewport() positions the viewport with respect to the full screen coordinates at its upper left corner, 0,0. So we position the viewport relative to the overall screen but construct our graphics within the port to its own coordinates regardless of where the port will be positioned. I have found this confusing, which is my reason for the emphasis. The "clip" digit in the function call will terminate any line within the viewport at its borders if its value is other than zero.

We can position the same viewport at multiple locations throughout the screen. We can also create and display more than

```
/* RECT1.C
** Example for floodfilling with the four palettes.
** using the TURBO C Ver. 2.0 library routines.
*/

  /* ** 1st rectangle(20,20,300,50) ** */
   setlinestyle(0,0,3);
    setcolor(1);        /* color 1 border */
     rectangle(20,20,300,50);

  /* ** floodfill rectangle ** */
   setfillstyle(SOLID_FILL,2);  /* color 2 fill */
     floodfill(24,24,1);

  /* ** 2nd rectangle(20,53,300,80) ** */
   setlinestyle(0,0,3);
    setcolor(2);        /* color 2 border */
     rectangle(20,53,300,80);

  /* ** floodfill rectangle ** */
   setfillstyle(SOLID_FILL,1);  /* color 1 fill */
     floodfill(24,56,2);

  /* ** 3rd rectangle(20,83,300,110) ** */
   setlinestyle(0,0,3);
    setcolor(3);        /* color 3 border */
     rectangle(20,83,300,110);

  /* ** floodfill rectangle ** */
   setfillstyle(SOLID_FILL,2);  /* color 2 fill */
     floodfill(24,86,3);

  /* ** 4th rectangle(20,113,300,140) ** */
   setlinestyle(0,0,3);
    setcolor(2);        /* color 2 border */
     rectangle(20,113,300,140);

  /* ** floodfill rectangle ** */
   setfillstyle(SOLID_FILL,3);  /* color 3 fill */
     floodfill(24,116,2);

      getch();
       closegraph();
}
```

Listing 2. Illustrating the four color palettes availible with Turbo C version 2.0.

```
/* PLANE.C
** A drawing program using viewports.
** Using the TURBO C Ver. 2.0 library routines.
*/

/* graphmode = 2 this program for red aircraft */
/* given in heading.c  */

  /* ** define viewport ** */
   setviewport(0,0,50,50,1);

  /* ** draw fuselage as linked line segments ** */
   setlinestyle(0,0,1);  /* set style and width */
    setcolor(3); /* brown border */
     moveto(8,23);    /* start segment a */
      lineto(11,23);
     moveto(11,22);   /* start segment b */
      lineto(21,22);
     moveto(21,21);   /* start segment c */
      lineto(40,21);
     moveto(40,22);   /* start segment d */
      lineto(44,22);
     moveto(44,23);   /* start segment e */
      lineto(48,23);
     moveto(48,24);   /* start segment f */
      lineto(50,24);
     moveto(8,27);    /* start segment g */
      lineto(11,27);
     moveto(11,28);   /* start segment h */
      lineto(21,28);
     moveto(21,29);   /* start segment i */
      lineto(40,29);
     moveto(40,28);   /* start segment j */
      lineto(44,28);
     moveto(44,27);   /* start segment k */
      lineto(48,27);
     moveto(48,26);   /* start segment l */
      lineto(50,26);
     moveto(50,24);   /* start segment m */
      lineto(50,26);
     moveto(8,23);   /* start segment ag */
      lineto(8,27);
    setfillstyle(SOLID_FILL,2); /* red */
   floodfill(14,25,3);

  /* ** draw cockpit as five line cluster ** */
  /* ** solid line, 1 pixel thick ** */
    setcolor(1);  /* green */
     moveto(30,23);
      lineto(36,23);
     moveto(29,24);
      lineto(37,24);
     moveto(28,25);
      lineto(38,25);
     moveto(29,26);
      lineto(37,26);
     moveto(30,27);
      lineto(36,27);

  /* ** outline the upper and lower wings ** */
    setcolor(1);  /* green */
     moveto(21,21); /* start segment r */
      lineto(21,0); /* start segment s */
       linerel(19,21);
     moveto(39,20); /* start segment v */
      lineto(21,20);
     moveto(21,29); /* start segment t */
      lineto(21,50);/* start segment u */
       linerel(19,-21);
     moveto(39,30); /* start segment w */
      lineto(21,30);
     setfillstyle(SOLID_FILL,2); /* red */
      floodfill(24,10,1);
      floodfill(24,40,1);

  /* ** outline the upper and lower tail fins ** */
    setlinestyle(0,0,1);
     moveto(0,14);      /* start segment np */
      lineto(0,36);
       linerel(7,-10);  /* draw segment q */
     moveto(7,27);      /* start segment qo */
      lineto(7,23);
     moveto(7,23);      /* start segment o */
      lineto(0,14);
     setfillstyle(SOLID_FILL,2); /* red */
      floodfill(2,20,1);

  /* ** draw vertical fin as 3-pixel line ** */
    setlinestyle(0,0,1); /* still green */
     moveto(0,25);
      lineto(12,25);

      getch();
       closegraph();
}
```

Listing 3. The aircraft of Figure 2.

```
/* VIEWPORT.C
** Multiple viewport example with two objects
** using the TURBO C Ver. 2.0 library routines.
*/

/* graphmode = 2 this program  */
/* given in heading.c  */

  /* ** begin viewport repeat ** */
  /* ** define viewport ** */
   while(i--) {
```

```
    setviewport(left_col,top,rite_col,bottom,1);
     draw_poly();
        left_col += 10; top += 10;
        rite_col += 10; bottom += 10;
   setviewport(left_col,top,rite_col,bottom,1);
     draw_circ();
        left_col += 10; top += 10;
        rite_col += 10; bottom += 10;
                }
      getch();
       closegraph();
}

/* == draw and fill rhombus == */
draw_poly()
{
int rhombus[] = { 0,0, 10,0, 7,8, 3,8, };

  /* ** draw and fill the rhombus ** */
   setlinestyle(0,0,1);
    setcolor(2);   /* red  border */
     setfillstyle(SOLID_FILL,2);  /* red fill */
      fillpoly(sizeof(rhombus)/(2*sizeof(int)), rhombus);
}

/* == draw and fill circle == */
draw_circ()
{
int x = 5, y = 5, r = 4;

  /* ** draw and fill the circle ** */
   setcolor(3);    /* brown border */
    circle(x,y,r);
     setfillstyle(SOLID_FILL,1);  /* green fill */
      floodfill(5,5,3);
}
```

Listing 4. A program illustrating multiple viewports.

```
/* TANK.C
** Left directed stick line tank figure.
** using the TURBO C Ver. 2.0 library routines.
*/

/* graphmode = 0 this program  */
/* given in heading.c  */

  /* ** begin animation ** */
  /* ** define viewport ** */
   while(i--) {
    setviewport(left_col,189,rite_col,199,1);
     draw_tank();
      delay(20);  clearviewport();
        left_col -= 5; rite_col -= 5;
              }
     draw_tank();
      getch();
       closegraph();
}

/* == draw tank as sequence of horiz lines == */
draw_tank()
{
    setlinestyle(0,0,1);  /* solid line, one pixel wide */
    setcolor(2);          /* tank top is red */
     moveto(9,0);  lineto(15,0); /* segment a */
    setcolor(3);          /* tank gun is brown */
     moveto(1,2);  lineto(7,2);  /* segment b */
    setlinestyle(0,0,3);  /* solid line, 3 pixels wide */
    setcolor(2);          /* tank top is red */
     moveto(7,2);  lineto(17,2); /* segment c */
    setlinestyle(0,0,1);  /* solid line, one pixel wide */
     moveto(9,4);  lineto(15,4); /* segment d */
    setcolor(1);          /* lower tank is green */
     moveto(5,5);  lineto(19,5); /* segment e */
     moveto(4,6);  lineto(19,6); /* segment f */
     moveto(5,7);  lineto(19,7); /* segment g */
     moveto(6,8);  lineto(18,8); /* segment h */
     moveto(7,9);  lineto(17,9); /* segment i */
}
```

Listing 5. A program illustrating animation with a small object.

one viewport at a time as well. This is shown in Listing 4. In this program two objects differing in shape and colors are distributed in a downward slant from the screen origin. Two library drawing functions, fillpoly() and circle() are used for the object creations. Note that each object is formed in its own function call. We can create as many objects as we like and use control logic to call them as appropriate.

Viewports offer an excellent approach to animation. The aircraft of Listing 3 is easily "flown" across the screen by embedding the viewgraph in a while loop incorporating variables for the position and the function cleardevice() to briefly blank the object. (Because of its size the aircraft tends to "flop" rather than fly!) Listing 5 illustrates animation with the small tank object (Figure 3) created by a series of colored parallel lines. Animation with a small object of this kind is very rapid. Exciting screen action games are possible through the combination of a background setting and animated objects in viewports.

Summary

We have learned that the IBM graphics system is uniquely distinct in almost every respect from the text mode. The distinctions arise from hardware considerations. The variety of graphic adapters, mostly downward compatible, contribute to the difficulty of creating widely portable graphics programs. The advances in library functions make it possible to create C graphics programs previously possible only with assembly. Screen graphics design is made easier by the use of a grid scaled in screen coordinates. Program examples illustrating object drawing, viewport manipulation, and animation show many of the graphic possibilities while providing understanding in how to use the library functions for drawing, coloring, and positioning our objects.

References

The following are excellent sources of related information.

TURBO C Version 2.0 Reference Guide, Borland International, 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95066-0001

Roger T. Stevens, *Graphics Programming In C*, M&T Publishing, Inc., 501 Galveston Drive, Redwood City, CA 94063

Nabajyoti Barkakati, *The Waite Group's Turbo C Bible*, Howard W. Sams & Co., 1989

User Disk

The code from *Standardize Your Floppy Disk Drives* and *Graphics Programming With C On the IBM PC* are available on a 5.25" 360K PC format disk for $10 postpaid in the US.

Lazy Evaluation

by Marla Bartel, Hawthorne Technology

You probably thought that an article on lazy evaluation would be about a technique for doing a product review without much effort. You know, the kind some magazines do so they can mention a hot new product, but manage not to tell you anything useful about the product. This article is not about lazy authors. What we are presenting here is the advantage of letting your computer be lazy while getting it's job done. It is wise to avoid work that does not need to be done, even if it is a machine doing the work.

People who write compilers put a lot of effort into making their compiler into one that will produce fast code. One method of making code that will run faster is a technique known as lazy evaluation. This technique cuts the amount of work the computer has to do to evaluate a logical expression. In a complex logical expression you can often know the result of the expression without evaluating all of it. By ending the evaluation as soon as the result is known, the computer does not waste time completing the expression when there is nothing to be gained.

Some languages allow the programmer to specify whether or not all of the conditions in an expression are going to be tested or if only a minimal number of conditions will be tested. Languages like ADA and C have separate operators for lazy and non-lazy evaluations. Other languages, like FORTRAN and PL/M allow the programmer to specify lazy evaluation by the level of code optimization attempted. Then there are the languages like BASIC and Pascal that don't offer any control over the way the evaluation is handled. Most Forths don't have any built in means of doing lazy evaluation but it is a feature that can be added. I am going to show you how to add this feature to your Forth and some ways of using it.

Lazy evaluation is not a new idea. It is used by every experienced electronics technician. You check to see if a circuit board has proper power before you bother checking for signal levels at the chips. If the voltages you need aren't there, you don't test any further (at least until you solve that problem). Not making this limited test in the first case could damage your test equipment or at least waste your time. You can think of it as a triage system for your computer to use on software. You let the computer spend its time working on problems where the outcome is unknown. Once you know how a logical evaluation will end, stop working on it.

You can use lazy evaluation as a programmers' safety net. Example 1 contains multiple compares, an arithmetic expression, and a logical AND. In some cases, processing a complete expression without checking each comparison not only wastes time but could have side effects. In this example, if the first condition fails the remaining parts may produce a run time error such as a divide by zero. Have you ever seen some other obviously less skilled programmer than yourself write code that inadvertently causes a divide by zero? MS-DOS feels this is sufficient to require locking up the machine and waiting for it to be rebooted, (which at times is inconvenient).

```
Example 1.      Divide By Zero

IF A <> 0 AND ( XYZ / A ) < 14 THEN  ----
```

In example 1, using a language with non-lazy evaluation, if A=0 the calculation of XYZ/A results in a divide by zero. Using lazy evaluation, when the A<>0 test proved to be false, the remainder of the tests would be abandoned.

A series of tests ORed together, (A OR B OR C), will be true as soon as one of the tests is found to be true. In the same way, the series of tests ANDed together (A AND B AND C), is false as soon as one of the tests is found to be false.

In the normal evaluation of a Forth expression that involves ANDing the result of several comparisons, each comparison is done and all of the logical operations are processed. This takes time and stack space. The lazy way to implement lazy evaluation techniques with your Forth is by adding two words, FEXIT (false exit) and TEXIT (true exit). At each level of your evaluation you do a conditional exit if the result can be determined at this point.

```
Example 2.
Lumber Mill Saw Startup
check_log_align FEXIT       check_saw_clear FEXIT
check_safety_switch FEXIT   saw_power_on  make_cut
; -- test conditions and abort if not ready
```

Example 2 is a pseudo code example of how you could use FEXIT. This is typical of a test that could be used as a startup sequence in many industrial applications. This is not an example that I have any personal experience with but I have vivid images of it from discussions at the GOFIG (Greater Oregon Forth Interest Group) meetings I attended. There are quite a few computers dedicated to turning trees into lumber. I guess the guys in the cutting room get a bit upset if your computer program causes a saw blade to break. At the speeds these blades are traveling, they explode. Not very safe. Getting back to the example, if the log is not lined up for the cut that is required, there is no reason to continue. The cut sequence is aborted. If the saw blade is obstructed you don't want to start it turning. Then you check the safety switch which indicates the operator is behind his OSHA safety shield. If any one of these requirements had not been met, the sequence would not have continued. Now you can turn on the saw and make the cut.

Another use for TEXIT and FEXIT is in control applications. If something needs to be done in different ways at different times, different procedures are tried and as soon as one of them works the routine is finished. These words can also be useful in searching or lookup. The search is stopped as soon as a match is found.

The words TEXIT and FEXIT can be used to create words that do a lazy evaluation of logical expressions. These words can be used to create special words that perform a test. They can also be used as part of a more complex word.

To use these words to construct a lazy evaluation create a special word for the test. Then use TEXIT for an OR expression and FEXIT for an AND expression. If the word is exited the result is left on the stack. If the exit is not taken the intermediate result is removed from the stack to clean it up since it will no longer serve a useful purpose by being there. Any word that returns a TRUE or FALSE result can use this method.

This is how the words TEXIT and FEXIT can be added to most versions of Forth. In HT-FORTH these words are compiled macros so while they work like the ones shown here, they don't look the same as these. You may have to make some small

changes so these examples will work in your version of Forth. Don't use these words inside of a DO LOOP because the DO LOOP in most versions of Forth uses the return stack to store the loop parameters so the return address is buried. ENDIF is used instead of THEN because we like it better.

Lazy Forth Words

When these words find the desired result, they drop one level of stack and no return is done. When returning with a result they don't drop any part of the stack.

TEXIT – true exit

Description: Exit from a word if the top of the stack is TRUE. If the exit is taken, the TRUE flag is left on the top of the stack. If the exit is not taken the FALSE flag is removed from the stack.

```
: TEXIT   ( T -- T )  ( F -- )
  IF TRUE R> DROP ENDIF
  ;
```

Example 3 will return the value of:

```
(N=3) OR (N=5) OR (N=7) OR (N=9)
```

Example 3a shows the lazy way and 3b is the standard way of evaluating the value of this expression in Forth. Notice that in example 3a there is no special action after the last test. The final test will leave the TRUE or FALSE on the stack and there is nothing to gain by exiting the word rather than just continuing to the semi-colon.

```
Example 3a  lazy
: TXTST  ( N -- N  tf )
  DUP 3 = TEXIT    DUP 5 = TEXIT
  DUP 7 = TEXIT    DUP 9 =
  ;

Example 3b  standard
: TXTST  ( N -- N  tf )
  DUP  3 =       OVER 5 = OR
  OVER 7 = OR    OVER 9 = OR
  ;
```

While the code for example 3a looks as long as that for 3b, your computer won't have to go through all of it every time as it would with the code in example 3b.

FEXIT – false exit

Exit from a word if the top of the stack is FALSE. If the exit is taken, the FALSE flag is left on the top of the stack. If the exit is not taken the TRUE flag is removed from the stack.

```
: FEXIT   ( F -- F ) ( T -- )
  NOT IF FALSE R> DROP ENDIF
  ;
```

Example 4 will return the value of:

```
(N>4) AND (N<30) AND (N<>7) AND (N<>9)
```

Notice that there is no special action after the last test. The last test leaves TRUE or FALSE on the stack and ends the word through the semi-colon.

```
Example 4a  lazy
: FXTST  ( N -- N  tf )
  DUP 4  >  FEXIT   DUP 30 < FEXIT
  DUP 7 <>  FEXIT   DUP  9 <>
  ;

Example 4b  standard
: FXTST  ( N -- N  tf )
  DUP  4  >        OVER 30  < AND
  OVER 7 <> AND    OVER 9  <> AND
  ;
```

With these two words added to your Forth you can duplicate some of the standard functions of C. Here are a few you might want to have around.

ISALPHA tests the item that is on the top of the stack to see if it is an ASCII alphabetic character. You exit false if the number on the stack is lower than any alphabetic values in the ASCII table: A = 65 ... Z = 90 ... a = 97 ... z = 122. If the number is within the range of upper case characters you can exit true. If you didn't exit here you know the number is not upper case. At this point, if the number is less than lower case 'a', you can exit false. Now if it is then less than or equal to lower case z you exit true, otherwise you know the number is not an alphabetic character and the result of ISALPHA is false. Using lazy evaluation, if the first test had failed you would not have had to read this whole description.

```
: ISALPHA     ( a -- a tf )
  DUP  65  < FEXIT
  DUP  91  < TEXIT
  DUP  97  < FEXIT
  DUP  123 <
  ;
```

ISDIGIT checks to see if the item on the top of the stack is an ASCII digit (0 - 9). If it is less than zero, the false exit will be taken. If it is greater than zero, it is checked to see if it is greater than 9. If it is within range the top of the stack will indicate true.

```
: ISDIGIT     ( a -- a tf )
  DUP  48  <  FEXIT
  DUP  58  <
  ;
```

ISSPACE determines whether or not the character on the top of the stack is a white-space character. First it is checked to see if it is an ASCII space. If it is, you are done. The result is true. If not, you proceed to check for a TAB, a LINE FEED, and a CURSOR RETURN. If any of these turn out to be true, the exit will be taken and no further tests will be made.

```
: ISSPACE     ( a -- a tf )
  DUP  32  =  TEXIT
  DUP   9  =  TEXIT
  DUP  10  =  TEXIT
  DUP  13  =
  ;
```

ISLOWER is a word that checks to see if the item on the top of the stack is a lower case alphabetic ASCII character. First it checks to see if the number is too small. The exit is taken if it is. If the exit is not taken, the value is checked to see if it is within the range of the lower case characters. The result of this check will set the true/false result on the stack.

```
: ISLOWER     ( a -- a tf )
  DUP  97  <  FEXIT
  DUP  123 <
  ;
```

ISUPPER is like ISLOWER except it checks to see if the item on the top of the stack is an upper case alphabetic ASCII character. First it checks to see if the number is too small. If the exit is not taken, the number is checked to see if it is within the range of upper case characters.

```
: ISUPPER     ( a -- a tf )
  DUP  65  <  FEXIT
  DUP  91  <
  ;
```

By adding a couple new words to your Forth from time to time, you can continue to improve your Forth code. If after having a word for a while, you find that you don't use it enough to justify the space it takes, you can always take it out again. Try TEXIT and FEXIT. They give you a clean way to get out of evaluating a complete expression when it is not necessary. Some people think it makes Forth a little easier to understand. •

S-100

There's Still Life in the Old Bus

by Michael Broschat

In issue #35, I discussed the building of an S-100 EPROM burner board kit sold by Digital Research of Texas. I detailed how the motivating factor had been to change the way my Sierra Data Sciences computer was booting itself into some fancy serial interrupts, thereby possibly keeping a new disk controller board from operating correctly in that system. The newly-burned boot PROM in place, I booted the system successfully, only to discover that the new configuration made no difference. I now believe the real answer to have been that since the board operates completely within IM2 (Zilog's interrupt mode 2), where all I/O devices, including the disk controller, are interrupt-driven in the approved daisy-chain manner, there is no way the non interrupt-driven new floppy board could work reliably (without extensive understanding and modification of software driving both systems). Anyway, the whole question is moot. My SDS board died some months ago, never to be resuscitated.

So, what to do with a brand new EPROM burner? A couple things entered my already cluttered mind. Let me digress. Unlike Rick Charnes, to whom I offer my apologies, I have been less than satisfied with my Qume 102 terminal. When I first got it, the key action bothered me. Now I am used to it, and all others seem inferior. But that's an old story. The biggest problems have been: the poor way it handles video attributes (except for normal/dim), where it introduces a space for every new attribute, making all but the normal/dim feature useless for most of the terminal-dependent software that has been such a plus for the Z System; and its poor handling of data streams at 9600 bps and up. The terminal is capable of 19.2K operation, but even at 9600 it has had trouble since I got it. Very curious, that. The universal problem has been MicroPro installation programs (not the programs themselves). From much to most of the screen text is unreadable, a problem I later discovered has to do with dropped characters. One of the installation programs requires slowing the terminal down to 2400 baud before all text can be read. A couple years ago I went so far as to desolder the static RAM chips, which I learned were quite slow (about 450 ns, I think), replaced them with socketed (just in case) faster chips, and then discovered that none of this made the slightest difference. Qume responded to one inquiry with a ROM update (probably making it equivalent to Rick's more recent model), but that made no difference, either.

So, I learned how to add hardware handshaking (easy via the SDS Zilog SIO and a cable modification) and changed the default operating speed to 19.2K. That works perfectly, but of course not at a real 19.2K because of the handshaking.

I recalled seeing an ad in an old hacker magazine. This turned out to be *MICROSYSTEMS* and the later *MICROSYSTEMS/JOURNAL*. A company called Simpliway offered S-100 board kits for a couple different devices, one of which was a video board with all kinds of neat features. All those features meant one thing to me: fast. So, fresh from kit-building success, I decided to see whether Simpliway was still available. They are. Dan Lurey has kept up the front, at least, and answered my inquiry with a brochure. You can request your own, but just looking at the brochure was exciting. The bareboard video terminal is only $50 (you find and buy all the components), and there are various options up to supplying you with all the parts. After some thought and a conversation with Dan, I bought both the video board and a buffered I/O board, plus a few of the rarer parts. I have now finished the video board and want to tell you about it.

First, the manual is excellent. Much care has been taken to keep the non-expert in mind. The only real trouble I recall experiencing was in the finding of all the components, or more commonly, ones that would fit. I probably saved some bucks buying the parts myself, but it took a good 2-3 weeks before I got everything in a usable form.

The board uses an on-board Z80 to handle the actual work of driving the CRT chip (an Intel 8275). PROMs (you knew they had to get in here somewhere) hold the character definition code and of course the program that drives the whole works. It comes with a program that emulates a Televideo 910, but you are free to alter it in any way you like. I had a vision of changing the inevitable 8080 code into hot stuff Z80 code, but Roger Vilmur, evidently the programmer there, has done a fine job already. I might end up adding a function or two (haven't noticed a screen-blanking function), but the program looks solid. And, attributes don't take up space! And, it operates through the bus, either port addressed or memory mapped, at effective speeds plenty fast enough to satisfy me (another board's specs claim 80,000 baud, but although Simpliway prefers 19.2K, the actual speed is probably the same for each board--fast).

You need a regular old monitor and a keyboard. The monitor is easy—the VDB-A (its designation) allows both composite and separate video/synch connections, so IBM-type monitors can be used, but the keyboard is another story—you need a parallel one. I have ended up with an old Jameco no-function key, no-cursor key keyboard after missing out on a deal through Jameco itself on some snazzy Cherry full-featured keyboards. Perhaps you'll have better luck (I'm not through looking yet). Dan tells me he has designed an interface for standard IBM PC keyboards, but whether that will become available will probably depend on how much interest we can generate (count me in!).

Now, I'd love to tell you how wonderfully all this works, but I can't. The death

of my Sierra Data Sciences machine left me without a machine upon which to develop system software for any of the other controller boards I currently own (that story is worth a novel), so although the VDB-A is built and raring to go, there is nothing on which to run it—yet. Which leads me to the next discovery.

After a few failed experiments involving a borrowed machine (also interrupt-driven, which proved the final proof regarding the incompatibility of the aforementioned disk controller), I had to have a disk controller with a boot disk. Once again, the ads. I discovered what is probably the only S-100 disk controller manufacturer still in business (as an S-100 manufacturer)—Fulcrum of Healdsburg, California. They offer what proves to be a fine deal: their controller for $250, or with a WD1002 hard disk controller card for $300, or all that with a hard disk for $400. I took the latter. I never dreamed I'd own a hard disk, and actually had not looked forward to one, since their noise is more than I am comfortable with, coupled with the fact that I have also built a 1-meg RAM disk (also from Digital Research/Texas). But it was a bargain and an opportunity. Don't let either go by easily.

This disk controller (I had earlier bought their 8 MHz CPU board, which I was going to use with one of the other controllers I have) was quite a find, a truly pleasant shock, the discoveries about which came after receiving it. I wasn't prepared for the fact that it supports floppies, hard disks, and RAM disks without any sacrifice in system size! My Sierra Data Sciences system came with a 58K system. Although there was a 1K hole up there into which I plugged minimal versions of first ZCPR 3.0, then 3.3, most of the system was going to waste as it included hard disk drivers I couldn't use, features I couldn't disable, etc. I was stuck with 58K max, and anything I wanted to add would take it down further. This Fulcrum OmniDisk system is a full 64K in size, and there appears to be enough room left in the BIOS to fit Bridger Mitchell's DateStamper module. The OmniDisk accomplishes this miracle by off-loading almost all normal BIOS activity to the microprocessor on-board the disk controller. About the only thing the native BIOS does is make "system" calls to its partner. And data transfer is by DMA (or not, as you choose).

Off-loading all that processing to the disk controller allows for a virtually unlimited warehouse of functions. One of special use to me allows one to configure any floppy drive to one of the formats included within the configuration utility. A miracle here is that one, for the California Computer System, fits my SDS disks. I thought all those disks were lost for good. Throughput is also increased through full-track buffering. Each time a sector is requested, its whole track is read and the contents kept in RAM (right, on the OmniDisk). If the next sector needed is already in the RAM, well, no more reads.

The combination of this 8 MHz CPU and this truly remarkable disk controller makes for one ideal Z System machine. The full 64K system means that any of the several RSX-type add-ons that have been or will be developed for former CP/M machines (Backgrounder ii, DOSDisk, BYE, etc.) will be allowed to run without seriously affecting one's normal programs. That certainly cannot be said of my Sierra Data system. Now, for example, when I invoke WordStar 4.0 on the native Fulcrum system, WS4 tells me I have 59K of TPA; after running NZCOM, it tells me I have 55K; after running BGii on top of all that, I still have 52.3—plenty enough even for memory-hungry WordStar. These figures are before any kind of customization, so perhaps even more can be squeezed out. I am anxious to try Bridger's DOSDisk on this machine, since the off-board BIOS already contains the definitions for MS-DOS formats. Presumably, very little work will be necessary to get DOSDisk to read and write MS-DOS formatted disks on this machine (Fulcrum provides a Pascal transfer program to move data back and forth between the two incompatible formats, but DOSDisk allows the MS-DOS drive to be used transparently on a CP/M system).

Fulcrum has expressed willingness to release the system source code (you get the BIOS anyway, and even the CPU board monitor ROM contents), that is, the code that is driving the on-board 8085 (yeah, too bad it isn't a Z80). I haven't seen this yet, but it occurs to me that at least a couple different products could be derived from this: a system that puts the BDOS and BIOS segments (anything else?) into RAM above 64K (like any modern S-100 board, the OmniDisk and MPU-Z CPU board address 16 megabytes); and different ROMs for the OmniDisk, where an enterprising programmer could conceive and execute even more functions than are currently available for this system (perhaps even combining both these points). Anyway, even if nothing else happens, I'm happy. As I write this, I am using a "loaner" (from a former Fulcrum owner near my area--thanks, Tim) while a buggy chip on one of the boards is diagnosed and replaced, but as soon as I am running my own hardware I will at last be able to quit fooling with all this hardware stuff and get down to some serious Z System programming. But then there is that EPROM burner just waiting for more action... •

Companies Mentioned

Simpliway Products Company
P.O. Box 601
Hoffman Estates, IL 60195

Fulcrum Computer Products
459 Allan Ct.
Healdsburg, CA 95448
(707) 433-0202

Plu*Perfect Systems == World-Class Software

BackGrounder ii **$75**

Task-switching ZCPR34. Run 2 programs, cut/paste screen data. Use calculator, notepad, screendump, directory in background. CP/M 2.2 only. Upgrade licensed version for $20.

Z-System **$69.95**

Auto-install Z-System (ZCPR v 3.4). Dynamically change memory use. Order **Z3PLUS** for CP/M Plus, or **NZ-COM** for CP/M 2.2.

PluPerfect Writer **$35**

Powerful text and program editor with EMACS-style features. Edit files up to 200K. Use up to 8 files at one time, with split-screen view. Short, text-oriented commands for fast touch-typing: move and delete by character, word, sentence, paragraph, plus rapid insert/delete/copy and search. Built-in file directory, disk change, space on disk. New release of our original upgrade to Perfect Writer 1.20, now for all Z80 computers. On-disk documentation only.

ZSDOS ..... **$75, for ZRDOS users just $60**

Built-in file DateStamping. Fast hard-disk warmboots. Menu-guided installation. Enhanced time and date utilities. CP/M 2.2 only.

DosDisk ..... **$30 - $45**

Use MS-DOS disks without copying files. Subdirectories too. Kaypro w/TurboRom, Kaypro w/KayPLUS, MD3, MD11, Xerox 820-I w/Plus 2, ON!, C128 w/1571 -- $30. SB180 w/XBIOS -- $35. Kit -- $45. Kit requires assembly language expertise and BIOS source code.

MULTICPY **$45**

Fast format and copy 90+ 5.25" disk formats. Use disks in foreign formats. Includes DosDisk. Requires Kaypro w/TurboRom.

JetFind **$50**

Fastest possible text search, even in LBR, squeezed, crunched files. Also output to file or printer. Regular expressions.

To order: Specify product, operating system, computer, 5 1/4" disk format. Enclose **check**, adding $3 shipping ($5 foreign) + 6.5% tax in CA. Enclose invoice if upgrading BGii or ZRDOS.

Plu*Perfect Systems
410 23rd St.
Santa Monica, CA 90402
(213)-393-6105 (eves.)

BackGrounder ii ©, **DosDisk** ©, **Z3PLUS** ©, **PluPerfect Writer** ©, **JetFind** ©
Copyright 1986-88 by Bridger Mitchell.

Advanced CP/M

Please Pass the Parameters

by Bridger Mitchell

In day-to-day coding the Z80 programmer calls subroutines using the Z80 registers to pass parameters and results to and fro. This mechanism is effective, fast, and generally transparent. Probably well over 95 percent of applications are coded this way.

Occasionally, however, it's valuable to have a wider range of tools at our disposal. This issue's Advanced CP/M illustrates several other mechanisms for use in programs when space is at a premium, general-purpose functions are required, and complex error recovery must be managed.

Inline print function

I'll begin with a familiar example. Assume that somewhere in the program there is a "print_to_nul" routine that prints a string of characters, pointed to by the HL register and terminated by a binary zero, on the console. The ordinary way of calling this subroutine is:

```
        ld      hl,stringmsg
        call    print_to_nul
        call    next_function
        ; ...
stringmsg:
        db      'A string',0
```

The alternative inline_print function places the string "in line" -- right in the flow of the code itself:

```
        call    inline_print
        db      'A string',0
        call    next_function
;...
;
inline_print:
        ex      (sp),hl
        call    print_to_nul
        ex      (sp),hl
        ret
```

Instead of using the HL register to pass the address of the string, the inline_print function makes use of the return address that is already on the stack when it is called. This saves loading a register with the string address and may increase the readability of the source code by placing the message near its use. It is, however, a nuisance when debugging.

---

*Bridger Mitchell is a co-founder of Plu*Perfect Systems. He's the author of the widely used DateStamper (an automatic, portable file time stamping system for CP/M 2.2); Backgrounder (for Kaypros); BackGrounder ii, a windowing task-switching system for Z80 CP/M 2.2 systems; JetFind, a high-speed string-search utility; DosDisk, an MS-DOS disk emulator that lets CP/M systems use pc disks without file copying; and most recently Z3PLUS, the ZCPR version 3.4 system for CP/M Plus computers.*

*Bridger can be reached at Plu*Perfect Systems, 410 23rd St., Santa Monica CA 90402, or at (213)-393-6105 (evenings).*

Multiple parameters

Functions needing several parameters are often called by loading separate registers:

```
        ld      hl,param1
        ld      de,param2
        ld      bc,param3
        call    funct3
```

When more parameters are involved, one can also use the alternate registers, or the stack:

```
        ld      hl,param1
        push    hl
        ld      hl,param2
        push    hl
        ...
        call    function3
```

But as with the print_to_nul function, there's an in-line way to pass these parameters:

```
        call    inline_function3
        dw      param1
        dw      param2
        dw      param3
```

This may be worthwhile if 'function3' is called many times. And, if the function has a variable number of parameters, the number of parameters can be passed as the first argument:

```
        call    functionX
        dw      N
        dw      param1
        dw      param2
        ...     ...
        dw      paramN
```

Still more flexible is to use an intermediate dispatching routine. The calling sequence is then

```
        call    dispatcher
        dw      functionX
        dw      N
        dw      param1
        dw      param2
         ...
```

Inline register functions

When the same operation must be performed many times throughout a program, specialized routines may be able to reduce code size. The following example is based on a technique used in Mike Arenson's PMATE editor. There is a pool of 16-bit words (pointers, indexes, counts, and so forth) that require frequent arithmetic operations—most commonly comparisons. These words are kept together in a data pool of not more than 128 words, beginning at some base address.

To compare two words—word1 and word2—the calling sequence is simply:

Figure 1. Error handling in nested calls.

```
callit: exx
        pop     hl
        ld      e,(hl)          ; get 1st parameter
        inc     hl              ; (addr of error routine)
        ld      d,(hl)
        inc     hl
        push    de              ; push it onto stack
        ld      de,-1           ; push -1 sentinel
        push    de
        ld      e,(hl)          ; get 2nd parameter
        inc     hl              ; (addr of routine to call)
        ld      d,(hl)
        inc     hl
        push    hl              ; push the ret addr (following the params)
        ld      hl,normalret    ; push addr of normal termination routine
        push    hl
        push    de              ; push addr of function to be called
        exx
        ret                     ; 'call' it

; terminate ok
normalret:
        exx
        pop     hl              ; get ret address from stack
        pop     de              ; strip off sentinel (-1)
        pop     de              ; strip off error address
        push    hl              ; put ret address onto stack
        exx
        ret                     ; and continue there


; Get inline error code
; Clean up stack and dispatch to error handler
;
error:  exx
        pop     hl              ; point at argument
        ld      a,(hl)          ; get byte argument (error code)
        ld      de,1            ; prepare an addend of 1
popoff: pop     hl              ; pop and discard stack parameter
        add     hl,de           ; until -1 encountered (setting the CY)
        jr      nc,popoff       ; ..continue popping the stack
        exx
        ret                     ; return to error handler addr
;
```

```
        call    inline_compare
        db      R1, R2
```

R1 and R2 are the offsets for word1 and word2, in number of bytes, from the base of the data pool. Thus, if the data structure is:

```
base:   dw      word0
        dw      word1
        dw      word2
```

then

```
        R1      equ     (word1-base)
        R2      equ     (word2-base)
```

The inline_compare routine returns with the Z80 flags set according to whether word1 is less than, equal to, or greater than word2. The routine preserves all registers except the flags.

It's hard to do better than 5 bytes of overhead to call a 16-bit compare routine! If, however, a Z80 restart instruction is available for use, the call can be trimmed to just three bytes:

```
        RST     20h
        db      R1, R2
```

In addition, the application must initialize three bytes at 0020h to be:

```
        jp      inline_compare
```

Of course, I must immediately say that such code would not be fully portable, because some Z80 systems use the restart instructions for BIOS operations.

Error recovery

Assemblers, compilers, linkers and other high-powered tools must travel deeply nested paths as they parse, analyze and build up their intermediate and final data structures. It's a challenge to recover from an input error—a missing semicolon, a mistyped digit—and then proceed with processing to provide as much useful diagnostic output as possible.

A useful strategy is to establish an error-handler for each major type of problem, and have the program jump to it when trouble is detected. For example the C language function library generally provides the setjmp/longjmp pair of functions. The program calls setjmp to establish a "bailout" return point, and then later calls longjmp to exit from any subsidiary routine back to this point.

The following assembly-language example is based on the error-recovery code used in the TDL assemblers and linkers. A major subroutine, say the symbol parser, is called indirectly with two in-line parameters—the address of an error handler and the address of the primary function.

```
        call    callit
        dw      error_handler
        dw      parser
```

The parser itself calls many routines, and they of course call others, quickly leading to considerable nesting. Along the way there will be many validity checks. If an error is detected, the code calls a general-purpose error handler and passes an in-line parameter to indicate the type of error.

```
        call    error
        db      err_code
```

This error routine then typically reports the error code to the console with a suitable text message and then dispatches directly to 'error_handler' which cleans up memory, open files and so forth, and then resumes the program. But how does the general-purpose error routine know which error handler to use? And how does it get the stack correctly restored, since it contains a possibly large number of return addresses, and other parameters that separate the point of error and the original calling point?

The technique is to place an extra parameter on the stack, a sentinel, that will identify the error handler's address. In this case it is -1, or 0FFFF hex, a value that cannot occur as a return address in an application and (because of the parameter-passing conventions used in this application) also cannot be a parameter on the stack. The error routine simply pops the stack repeatedly until it finds the -1 value. The value above that is then the error_handler address.

The key routines are shown in Figure 1. In this case, the alternate Z80 registers are dedicated to processing the normal and error termination conditions so that the code can retain the main registers for returning results from subroutines.

Other techniques

One of the rewarding aspects of programming your own applications is that there are always several ways to accomplish the task. Program design involves examining the alternatives and choosing a suitable path. And frequently that process leads to refinements that yield more transparent and readily maintained code as well. I'm certain that a number of TCJ readers have encountered and extended other parameter-passing techniques. If you're one of them, drop me a note and we can continue the discussion in a future column. •

Real Computing

The National Semiconductor NS32032

by Richard Rodman

By now, you've probably heard: the 32764 has been canceled. The processor that was to be the 32764 will become a killer processor for the embedded systems market, with a performance goal of 90 to 100 MIPS. And remember, NS32 MIPS are real CISC MIPS.

The only thing I don't like about National's approach is their belief that embedded systems don't need an MMU. Needing an MMU has little to do with applications—it has to do with multitasking. Besides, the MMU is one of the best features of the NS32 architecture—simple, well thought out, clean, integrated.

And, on the subject of MMUs: Intel's 386 processor has a demand-paged MMU which is a blatant copy of the NS32 MMU. It uses 4K byte pages in a two-level scheme identical to that of the 32382. In fact, while all of the names have been changed, the 386's page table entries are *exactly* the same as the NS32382's, except that the A bit (R bit) and D bit (M bit) have been shifted up one bit. Of course, it isn't integrated into the instruction set like the NS32's, and the 386 has all of that awful segment selector garbage, so page fault processing will be really difficult. It does prove one thing, though: You can teach an old dog new tricks!

New NS32 Systems

National Semiconductor has an evaluation and development board available for their 32CG16 computer-graphics version of the 32016. This board includes the raster graphics chip set, and is an AT-form-factor motherboard with AT-bus slots. The part number is NSV-CG16-EDB, and it's available for $1195 from your local sales office.

Other people out there are working on PC motherboards with NS32 processors, similar to Peripheral Technology's 68000 board, but none are available yet. Watch this space for further information.

Some not-so-new ICM-3216 boards are available, as well as an S-100 board with a Z-80 and a 32016. If you're interested in these, leave a message on my BBS.

Listing 1

```
;ICUTEST.A32 - Test program for the ICU.

icubase .equ    0FFFE00h        ;base of ICU
mymod   .equ    0200h           ;copy of my module table entry
        .code
        br      init            ;s/b 2 bytes
        br      inthandler      ;interrupt handler
;Copy my module table to ''mymod''.  This is so that, when you load
;another program later, the module entry won't be overwritten by that
;for the new program.

init:
        movzwd  mymod,r1
        sprw    mod,r0
        movzwd  r0,r0
        movd    0(r0),0(r1)
        movd    4(r0),4(r1)
        movd    8(r0),8(r1)
;Set interrupt vector descriptors in place.
        sprd    intbase,r0              ;pick up current intbase
        movd    #00020200h,64(r0)       ;vector descriptor, vector 16
;Initialize ICU per initialization sequence shown in databook figure 3-10
;ICU clock on CPU-32016 is 2MHz.  Note that register number is doubled
;for CPU-32016.  For a PD-32, it would be quadrupled; for the 32532
;Designer Kit, it would be unchanged.
        movd    #icubase,r0
;Set MCTL leaving coutd at logic 1
        movb    #01000000b,32(r0)       ;MCTL
;Halt counters by clearing bits crunl and crunh in CCTL
        movb    #00000000b,44(r0)       ;CCTL - both counters stopped
;Initialize LCSV, HCSV and CIPTR
;Timer countdown = 20000 (100Hz, 10ms) = 4E20
        movb    #20h,48(r0)             ;LCSV - L-counter starting value
        movb    #4Eh,50(r0)
        movb    #00h,36(r0)             ;CIPTR - change clocks to vector 0
;Write counter starting value into LCCV to prevent long initial counts
        movb    #20h,56(r0)             ;LCCV - L-counter current value
        movb    #4Eh,58(r0)
;Initialize CICTL
        movb    #00000011b,46(r0)       ;CICTL - enable ints on L counter
;Initialize IPS, PDIR, OCASN, PDAT
        movb    #00000000b,40(r0)       ;IPS - all pins I/O
        movb    #11111111b,42(r0)       ;PDIR - all pins input
        movb    #00000000b,34(r0)       ;OCASN - no clocks on pins
;Initialize SVCT, ELTG, TPL, FPRT
        movb    #16,2(r0)               ;SVCT vector = 16
        movb    #0FFh,4(r0)             ;ELTG - all ints level-triggered
        movb    #0FFh,6(r0)
        movb    #0h,8(r0)               ;TPL - all ints low true
        movb    #0h,10(r0)
        movb    #0,28(r0)               ;FPRT - 0 first vector
;Reprogram MCTL COUTD bit to enable interrupts
        movb    #00000000b,32(r0)       ;MCTL
;Start counters
        movb    #01000100b,44(r0)       ;CCTL - only L counter running,
                                        ;not prescaled
;Initialize IMSK
        movb    #11111110b,20(r0)       ;IMSK - unmask vector 0
        movb    #11111111b,22(r0)
```

More Free Operating Systems

There are other free operating systems besides Bare Metal. By "free", I actually mean that they are available to anyone at low cost with complete source code. I'll briefly touch on a couple of these.

Xinu is described in the book *Operating System Design, the Xinu Approach* by Douglas Comer (Prentice-Hall, 1974). Source is available from Prentice-Hall on magnetic tape for PDP-11s, or from Austin Code Works for PCs. Xinu is a very simple multitasking operating system; it has educational merit, but as a real working environment, it's pretty bare-bones. The name "Xinu" stands for "Xinu Is Not Unix", and it isn't, not by a long shot.

Minix is described in the book *Operating Systems - Design and Implementation* by Andrew Tanenbaum (Prentice-Hall, 1987). Source and executables are available on diskette from Prentice-Hall for $79. "Minix" stands for "Mini-Unix". Actually, Minix is a pretty complete Unix version 7 implementation for PCs. If you have a PC but want Unix, get Minix. The book is also a good tutorial on how various things work in operating systems, too, although it has a serious weakness in that other implementations (non-Unix features) are not even mentioned.

Other operating systems that are available for free, with a commensurate amount of documentation and support, are Trix and Uzi. Trix is a Unix-like operating system for the 68000. Uzi is a Unix-like kernel for the Z-80.

```
;Set up for interrupts to happen
        setcfg  [i,f]                   ;make interrupts vectored
        bispsrw #0000100000000000b      ;enable interrupts
        rxp     0                       ;return to srm

;-- Interrupt handler --
;This simple handler increments a value at 50 hex consisting of:
;       50      - 1/100 seconds (10 ms)
;       51      - seconds
;       52      - minutes
;       53      - hours
;It could be expanded to keep time in other fashions if desired.

inthandler:
        save    [r0]
        movzbd  #50h,r0
        addqb   #1,0(r0)                ;incr hundredths of seconds
        cmpb    0(r0),#100
        blt     intdone
        movqb   #0,0(r0)
        addqb   #1,1(r0)                ;incr seconds
        cmpb    1(r0),#60
        blt     intdone
        movqb   #0,1(r0)
        addqb   #1,2(r0)                ;incr minutes
        cmpb    2(r0),#60
        blt     intdone
        movqb   #0,2(r0)
        addqb   #1,3(r0)                ;incr hours
        cmpb    3(r0),#24
        blt     intdone
        movqb   #0,3(r0)
intdone:
        restore [r0]
        reti

        end
```

But if I get to pick a "book of the month", like other columnists, the book of this month would be the new edition of Ted Nelson's *Computer Lib/Dream Machines* (Microsoft Press). While we have fulfilled the dream of making machines available to everyone, how little progress we have made toward making them easy to use! Cybercrud lives on.

Incidentally, I tried out a computer called the Macintosh, made by Apple Computer, Inc. While it has some direct manipulation features that are kind of nice, there's still a lot of mysterious, invisible features (like "cloverleaf D to duplicate"), and no visible way to request Help. There's a new concept called a "command line" that lets you make requests of the computer using English-like sentences. After all, some actions are not easily expressible in direct manipulation terms. I hope it catches on.

The System Architecture for the 90s

In other big news, DEC finally rolled out their long-anticipated VAX 9000 system. This machine features, besides many hardware innovations in packaging and interconnection, a new system architecture: the central switch architecture. The system should be able to achieve an I/O bandwidth far in excess of previous models that used the BI- or Q-buses. And as we know, I/O bandwidth is what really separates mainframes from PCs.

DEC also optimized its CISC processor so that the 80 percent of most-used opcodes execute in a single instruction cycle. Each processor can achieve 30 VAX MIPS. Currently, up to four processors can be installed. The central switch, called the System Control Unit (SCU), can set up four independent paths between these four processors and other system elements.

The central switch architecture represents the third stage in the evolution of bus architectures. Now, I'm not a bus expert, so forgive me if my jargon isn't correct.

In the first stage, the simple bus architecture, the processor signals on a single CPU board are converted to generic bus equivalents and used to communicate in a master-slave fashion with other boards in the system. Sometimes there can be multiple CPUs, but only one acts as a master at a time. This is the architecture of the S-100, PC, AT, NuBus, VME bus, Micro Channel, and nearly all other common buses. Some of these buses have elaborate protocols for supporting multiple CPUs on the bus, but the practical differentiating feature is that the bus has an address portion or phase and a data portion or phase.

The second stage is a message-passing bus. Unlike the simple bus architecture, every board in a message-passing bus system must be a complete system, with RAM memory, processor, clock, etc. that it needs. The separate processors communicate with each other in a message-passing fashion, similar to a local area network. Now the separate entities which sit on this bus might have their own local simple buses for expansion in a multiple bus system. The only commercial bus I know of with this architecture is Multibus II. As might be expected, this architecture imposes a greater burden in parts cost, but achieves greater performance because, when not communicating, the individual processors can run at full speed in parallel. However, when communicating, only one processor can "speak" on the bus at a time. Further, the speed of the bus is limited to the speed at which the slowest processor can "speak".

The central-switch architecture is similar to the message-passing architecture, except that the central switch is used to set up multiple independent paths between interconnected processors. Now, separate transfers between processors can take place simultaneously. Again, individual processors may have their own local

buses.

The central-switch architecture, while new to general-purpose computerdom, has actually been used in the telephone industry for years to create switches, which need tremendous I/O bandwidth but limited computing requirements. New hardware technologies, however, could make this system architecture possible for even small systems. Texas Instruments makes a part, the 74ACT8841, which implements a 16-port, 4-bit-wide switch allowing any port to be connected to any other port. Using 8 of these devices, 16 32532 systems could be interconnected to produce a small, but very high performance system.

Transputer fans have a C004 crossbar switch device, which is similar but switches the transputer's serial links. The general verdict on the transputer seems to be: "Wonderful hardware, but too hard to program."

And there is the true price of the central-switch system. Computers are developing slowly out of their tinker-toy stage and becoming genuinely difficult to program. People expect more and more of software, but the software tools of today are still crude. Yet better tools, once developed, are locked away as trade secrets or, even worse, software patents. Any future worth living in is worth making sacrifices for.

The NS32202 Interrupt Control Unit (continued)

Last time, I described the NS32202 ICU. This time I'm including a sample program to test out the operation of your ICU (Listing 1). This program is your typical interrupt demonstration that makes your system into a digital clock. Remember that, for your system, you may have to change the register number offsets to the register times 1, 2 or 4. The listing shows register times 2.

Next Time

Next time I'll have some benchmark data on the NS32532, since I have my Designer's Kit running now. •

Where to call or write

BBS: 703-330-9049 (Richard Rodman)

Austin Code Works
11100 Leafwood Lane
Austin TX 78750-3409

Prentice-Hall, Inc.
Route 59 at Brook Hill Drive
West Nyack, New York 10995

Peripheral Technology
1710 Cumberland Point Dr. #8
Marietta GA 30067

Technology Resources

K-OS ONE—Single user generic 68000 operating system for your 68000 hardware. It uses the MS-DOS disk format, and includes the operating system with source code (written in HTPL), an editor, assembler, and HTPL compiler. A sample BIOS code and a boot loader are included. This is **not** ready-to-run—you have to install the BIOS on your system, but the source code and language compiler are included **$50**

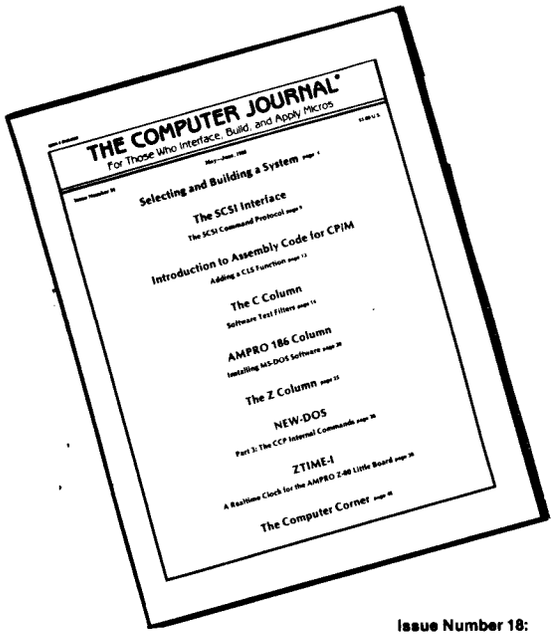
HT-Forth—A full featured, interactive Forth that works with the K-OS ONE operating system. It uses a full 32 bit stack and 32 bit arithmetic to take full advantage of the 68000. Programs are position independent and are limited in size only by the memory available. Source code compiles to inline macros, JSR, or BSR so there is no inner interpreter overhead. Standard ASCII files are used. Includes full screen editor and a Forth style 68000 assembler $100

68000Cross Assembler—Written entirely in 8086 assembly language, it is small and fast. All input and output is done with standard MS-DOS calls so it will run on any MS-DOS system, even those which are not totally PC compatible. All 68000 and 68010 instructions are supported. It has conditional assembly, the symbol table is in alphabetical order, and cross referencing is included. Include files are supported so it is easy to assemble big programs, but edit them in small pieces. An equate file can be produced for PROM based programming $50

ORDER FROM

Technology Resources
190 Sullivan Crossroad
Columbia Falls, MT 59912
Phone (406) 257-9119

Visa and Mastercard accepted
Prices postpaid in the U.S. and Canada



THE COMPUTER JOURNAL

Back Issues

Issue Number 1:

- RS-232 Interface Part One
- Telecomputing with the Apple II
- Beginner's Column: Getting Started
- Build an ""Epram"

Issue Number 2:

- File Transfer Programs for CP/M
- RS-232 Interface Part Two
- Build Hardware Print Spooler: Part 1
- Review of Floppy Disk Formats
- Sending Morse Code with an Apple II
- Beginner's Column: Basic Concepts and Formulas

Issue Number 3:

- Add an 8087 Math Chip to Your Dual Processor Board
- Build an A/D Converter for Apple II
- Modems for Micros
- The CP/M Operating System
- Build Hardware Print Spooler: Part 2

Issue Number 4:

- Optronics, Part 1: Detecting, Generating, and Using Light in Electronics
- Multi-User: An Introduction
- Making the CP/M User Function More Useful
- Build Hardware Print Spooler: Part 3
- Beginner's Column: Power Supply Design

Issue Number 6:

- Build High Resolution S-100 Graphics Board: Part 1
- System Integration, Part 1: Selecting System Components
- Optronics, Part 3: Fiber Optics
- Controlling DC Motors
- Multi-User: Local Area Networks
- DC Motor Applications

Issue Number 16:

- Debugging 8087 Code
- Using the Apple Game Port
- BASE: Part Four
- Using the S-100 Bus and the 68008 CPU
- Interfacing Tips & Troubles: Build a ""Jellybean" Logic-to-RS232 Converter

Issue Number 18:

- Parallel Interface for Apple II Game Port
- The Hacker's MAC: A Letter from Lee Felsenstein
- S-100 Graphics Screen Dump
- The LS-100 Disk Simulator Kit
- BASE: Part Six
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 1

Issue Number 19:

- Using The Extensibility of Forth
- Extended CBIOS
- A $500 Superbrain Computer
- BASE: Part Seven
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 2
- Multitasking and Windows with CP/M: A Review of MTBASIC

Issue Number 20:

- Designing an 8035 SBC
- Using Apple Graphics from CP/M: Turbo Pascal Controls Apple Graphics
- Soldering and Other Strange Tales
- Build a S-100 Floppy Disk Controller: WD2797 Controller for CP/M 68K

Issue Number 21:

- Extending Turbo Pascal: Customize with Procedures and Functions
- Unsoldering: The Arcane Art
- Analog Data Acquisition and Control: Connecting Your Computer to the Real World
- Programming the 8035 SBC

Issue Number 22:

- NEW-DOS: Write Your Own Operating System
- Variability in the BDS C Standard Library
- The SCSI Interface: Introductory Column
- Using Turbo Pascal ISAM Files
- The AMPRO Little Board Column

Issue Number 23:

- C Column: Flow Control & Program Structure
- The Z Column: Getting Started with Directories & User Areas
- The SCSI Interface: Introduction to SCSI
- NEW-DOS: The Console Command Processor
- Editing The CP/M Operating System
- INDEXER: Turbo Pascal Program to Create Index
- The AMPRO Little Board Column

Issue Number 24:

- Selecting and Building a System
- The SCSI Interface: SCSI Command Protocol
- Introduction to Assembly Code for CP/M
- The C Column: Software Text Filters
- AMPRO 186 Column: Installing MS-DOS Software
- The Z Column
- NEW-DOS: The CCP Internal Commands
- ZTIME-1: A Realtime Clock for the AMPRO Z-80 Little Board

Issue Number 25:

- Repairing & Modifying Printed Circuits
- Z-Com vs Hacker Version of Z-System
- Exploring Single Linked Lists in C
- Adding Serial Port to Ampro L.B.
- Building a SCSI Adapter
- New-Dos: CCP Internal Commands
- Ampro '186 Networking with SuperDUO
- ZSIG Column

Issue Number 26:

- Bus Systems: Selecting a System Bus
- Using the SB180 Real Time Clock
- The SCSI Interface: Software for the SCSI Adapter
- Inside AMPRO Computers
- NEW-DOS: The CCP Commands Continued
- ZSIG Corner
- Affordable C Compilers
- Concurrent Multitasking: A Review of DoubleDOS

Issue Number 27:

- 68000 TinyGiant: Hawthorne's Low Cost 16-bit SBC and Operating System
- The Art of Source Code Generation: Disassembling Z-80 Software
- Feedback Control System Analysis: Using Root Locus Analysis and Feedback Loop Compensation
- The C Column: A Graphics Primitive Package
- The Hitachi HD64180: New Life for 8-bit Systems
- ZSIG Corner: Command Line Generators and Aliases
- A Tutor Program for Forth: Writing a Forth Tutor in Forth
- Disk Parameters: Modifying The CP/M Disk Parameter Block for Foreign Disk Formats

Issue Number 28:

- Starting your Own BBS
- Build an A/D Converter for the Ampro L.B.• HD64180: Setting the wait states & RAM refresh, using PRT & DMA
- Using SCSI for Real Time Control
- Open Letter to STD-Bus Manufacturers
- Patching Turbo Pascal
- Choosing a Language for Machine Control

Issue Number 29:

- Better Software Filter Design
- MDISK: Adding a 1 Meg RAM disk to Ampro L.B., part one.
- Using the Hitachi HD64180: Embedded processor design.
- 68000: Why use a new OS and the 68000?
- Detecting the 8087 Math Chip
- Floppy Disk Track Structure
- The ZCPR3 Corner

Issue Number 30:

- Double Density Floppy Controller
- ZCPR3 IOP for the Ampro L.B.
- 3200 Hacker's Language
- MDISK: 1 Meg RAM disk for Ampro LB, part 2
- Non-Preemptive Multitasking
- Software Timers for the 68000
- Lilliput Z-Node
- The ZCPR3 Corner
- The CP/M Corner

Issue Number 31:

- Using SCSI for Generalized I/O
- Communicating with Floppy Disks: Disk parameters and their variations.
- XBIOS: A replacement BIOS for the SB180.
- K-OS ONE and the SAGE: Demystifing Operating Systems.
- Remote: Designing a remote system program.
- The ZCPR3 Corner: ARUNZ documentation.

Issue Number 32:

- Language Development: Automatic generation of parsers for interactive systems.
- Designing Operating Systems: A ROM based O.S. for the Z81.
- Advanced CP/M: Boosting Performance.
- Systematic Elimination of MS-DOS Files: Part 1, Deleting root directories & an in-depth look at the FCB.
- WordStar 4.0 on Generic MS-DOS Systems: Patching for ASCII terminal based systems.
- K-OS ONE and the SAGE: Part 2, System layout and hardware configuration.
- The ZCPR3 Corner: NZCOM and ZCPR34.

Issue Number 33:

- Data File Conversion: Writing a filter to convert foreign file formats.
- Advanced CP/M: ZCPR3PLUS, and how to write self relocating Z80 code.
- DataBase: The first in a series on data bases and information processing.
- SCSI for the S-100 Bus: Another example of SCSI's versatility.
- A Mouse on any Hardware: Implementing the mouse on a Z80 system.
- Systematic Elimination of MS-DOS Files: Part 2—Subdirectories and extnded DOS services.
- ZCPR3 Corner: ARUNZ, Shells, and patching WordStar 4.0

Issue Number 34:

- Developing a File Encryption System: Scramble data with your customized encryption/password system.
- DataBase: A continuation of the database primer series.
- A Simple Multitasking Executive: Designing an embedded controller multitasking system.
- ZCPR3: Relocatable code, PRL files, ZCPR34, and Type 4 programs.
- New Microcontrollers Have Smarts: Chips with BASIC or Forth in ROM are easy to program.
- Advanced CP/M: Operating system extensions to BDOS and BIOS, RSXs for CP/M 2.2.
- Macintosh Data File Conversion in Turbo Pascal.

Issue Number 35:

- All This & Modula-2: A Pascal-like alternative with scope and parameter passing.
- A Short Course in Source Code Generation: Disassembling 8086 software to produce modifiable assem. souce code.
- Real Computing: The National Semiconductor NS32032 is an attractive alternative to the Intel and Motorola CPUs.
- S-100 Eprom Burner: a project for S-100 hardware hackers.
- Advanced CP/M: An up-to-date DOS, plus details on file structure and formats.
- REL-Style Assembly Language for CP/M and Z-System: Part 1-selecting your assembler, linker, and debugger.
- ZCPR3 Corner: How shells work, cracking code, and remaking WordStar 4.0.

Issue Number 36:

- Information Engineering: Introduction
- Modula-2: A list of reference books
- Temperature Measurement & Control: Agricultural computer application
- ZCPR3 Corner: Z-Nodes, Z-Plan, Amstrand computer, and ZFILE!
- Real Computing: NS32032 hardware for experimenter, CPU's in series, software options
- SPRINT: A review
- ZCPR3's Named Shell Variables
- REL-Style Assembly Language for CP/M & Z-Systems, part 2
- Advanced CP/M: Environmental programming

Issue Number 37:

- C Pointers, Arrays & Structures Made Easier: Part 1, Pointers
- ZCPR3 Corner: Z-Nodes, patching for NZCOM, ZFILER
- Information Engineering: Basic Concepts; fields, field definition, client worksheets
- Shells: Using ZCPR3 named shell variables to store date variables
- Resident Programs: A detailed look at TSRs & how they can lead to chaos
- Advanced CP/M: Raw and cooked console I/O
- Real Computing: NS320XX floating point, memory management, coprocessor boards, & the free operating system
- ZSDOS-Anatomy of an Operating System: Part 1

Issue Number 38:

- C Math: Handling Dollars and Cents With C.
- Advanced CP/M: Batch Processing and a New ZEX.
- C Pointers, Arrays & Structures Made Easier: Part 2, Arrays.
- The Z-System Corner: Shells and ZEX, new Z-Node Central, system security under Z-Systems.
- Information Engineering: The portable Information Age.
- Computer Aided Publishing: Introduction to publishing and Desk Top Publishing.
- Shells: ZEX and hard disk backups.
- Real Computing: The National Semiconductor NS320XX.
- ZSDOS--Anatomy of an Operating System, Part 2.

Issue Number 39:

- Programming for Performance: Assembly Language techniques.
- Computer Aided Publishing: The Hewlett Packard LaserJet.
- The Z-System Corner: System enhancements with NZCOM.
- Generating LaserJet Fonts: A review of Digi-Fonts.
- Advanced CP/M: Making old programs Z-System aware.
- C Pointers, Arrays & Structures Made Easier: Part 3: Structures.
- Shells: Using ARUNZ alias with ZCAL.
- Real Computing: The National Semiconductor NS320XX.
- The Computer Corner.

Issue Number 40:

- Programming the LaserJet: Using the escape codes.
- Beginning Forth Column: Introduction.
- Advanced Forth Column: Variant Records and Modules.
- LINKPRL: Generating the bit maps for PRL files from a REL file.
- WordTech's dBXL: Writing your own custom designed business program.
- Advanced CP/M: ZEX 5.0—The machine and the language.
- Programming for Performance: Assembly language techniques.
- Programming Input/Output With C: Keyboard and screen functions.
- The Z-System Corner: Remote access systems and BDS C.
- Real Computing: The NS320XX
- The Computer Corner.

Issue Number 41:

- Forth Column: ADTs, Object Oriented Concepts.
- Improving the Ampro LB: Overcoming the 88Mb hard drive limit.
- How to add Data Structures in Forth
- Advanced CP/M: CP/M is hacker's haven, and Z-System Command Scheduler.
- The Z-System Corner: Extended Multiple Command Line, and aliases.
- Programming disk and printer functions with C.
- LINKPRL: Making RSXes easy.
- SCOPY: Copying a series of unrelated files.
- The Computer Corner.

Issue Number 42:

- Dynamic Memory Allocation: Allocating memory at runtime with examples in Forth.
- Using BYE with NZCOM.
- C and the MS-DOS Screen Character Attributes.
- Forth Column: Lists and object oriented Forth.
- The Z-System Corner: Genie, BDS Z and Z-System Fundamentals.
- 68705 Embedded Controller Application: An example of a single-chip microcontroller application.
- Advanced CP/M: PluPerfect Writer and using BDS C with REL files.
- Real Computing: The NS 32000
- The Computer Corner

TCJ ORDER FORM

Subscriptions		U.S.	Canada	Surface Foreign	Total
6 issues per year ☐ New ☐ Renewal	1 year	$16.00	$22.00	$24.00	
	2 years	$28.00	$42.00		
Back Issues		$3.50 ea.	$3.50 ea.	$4.75 ea.	
Six or more #'s		$3.00 ea.	$3.00 ea	$4.25 ea.	
				Total Enclosed	

All funds must be in U.S. dollars on a U.S. bank.

☐ Check enclosed ☐ VISA ☐ MasterCard Card # _____

Expiration date_____ Signature _____

Name _____

Address _____

City_____ State_____ ZIP _____

#43

THE COMPUTER JOURNAL

190 Sullivan Crossroad, Columbia Falls, MT 59912 Phone (406) 257-9119

Postscript which are indispensable for anyone working in that area. Other important titles are "Writing MD-DOS Device Drivers" by Lai, "Graphics Programming in Turbo C 2.0" by Ezzel, and "Programming in Clipper" (second edition) by Straley. I don't know if their titles are available by direct order, but get their catalog and check with them.

What other titles do you find useful? Who publishes them, are they still in print, and where can you get them? Send us the information, and we'll put it together into an article or a column.

EPROMs and Other Things

Broschat's article is very timely, because I am also delving into ROMs (mostly EPROMs) on various CPU, printer, and drive controller boards. I've long wanted to download the EPROM contents to disk where I could disassemble and revise them. Now I finally got a Periphco EPROM programmer running, but I still need an EPROM eraser.

EPROMs are the most frequently used non-volatile memory for developing embedded controllers, but the commonly available parts have been relatively slow (350 or 450 nanoseconds). This limits the system clock frequency to 2.0 or 2.5 MHz unless you insert wait states. CMOS parts, such as the 27C128-15 (16,384x8) with a speed of 150 nanoseconds (6 MHz) are available from Jameco or JDR for about $7, but they require a 12.5V programming pulse instead of the 25V pulse used with the older parts such as the 2732. My EPROM programmer only offers a choice between 21 and 25 volts, so it would have to be modified in order to support both 12.5 and 25 volt parts. The EPROM also have the disadvantage of requiring an intense short wave ultraviolet light for erasing. A decent EPROM eraser costs about $70.

Two alternatives to EPROMs are EEPROMs (Electrically Erasable Programmable Read Only Memory) and battery backed nonvolatile SRAM. The older EEPROMs required special voltages for programming and erasing, but newer devices need only 5 volts. The speed on EEPROMs has been running 250 to 350 nanoseconds, but faster devices are becoming available.

We carried an article in issue #1 on building an EPRAM (Erasable Programmable Random Access Memory) which described how to add battery back-up to CMOS SRAM which is now available with speeds of 100 nanoseconds and less. This is still a very viable idea for system development, but is too bulky for use in production units. Dallas Semiconductor (4350 Beltwood Parkway South, Dallas Texas 75244-3219) produces a nonvolatile SRAM with a built in battery which is very attractive for both development and production. Prototype samples are available from a special department for credit card orders (1-800-336-6933). I wish that other vendors made it so easy to order small quantities. Their DS1220 (2Kx8) is a reasonable $10.80. The DS1235 (32Kx8) is $51.57 which is a little steep. I don't have the specs on speed yet, but I have ordered a DS1220 for evaluation and I'll let you know how it works out.

The answer may be to use standard EPROM where the speed is acceptable and the changes are not fast and furious. Switch to faster 12.5 volt CMOS EPROMs where you need the speed (Texas Instruments has speeds down to 35 nanoseconds). Use battery backed CMOS RAM where you need both the speed and ease of changing the data.

I'll be building several EPRAMs for my own work, and I'll report more on them later. Issue #1 is still available for $3.50, and when they run out we'll supply a copy of the article for $1.50 (add $1.25 for foreign orders).

Incidentally, when I talk about part availability, I mean parts that I can order in one or two quantity today from Jameco, JDR, Digi-Key, Dallas, etc.—not what is only mentioned under new developments or available only in large quantity EOM direct orders.

Z-Fest 2 and Pieces of Eight

The Connecticut CP/M User's Group is in the process of scheduling the second Z-Fest, with a tentative date of May 19 at Trinity College. The date and place are not firm yet, but I need to call this to your attention in this issue because #44 will probably be too late (darn two month scheduling).

Check with *Pieces of Eight* c/o Lee Bradley, 24 East Cedar Street, Newington, CT 06111. I believe that up to date information will also be on the Eastern Z-Nodes.

They are to be commended in continuing to support those of us still interested in CP/M. Check with Lee about joining the group and receiving their newsletter. •

---

sonal programming language. That means you may earn your living doing C or Fortran, but when working on those fun or quick ten minute projects, out comes Forth and it is done. It means picking a package you like and adding your own personal features to help you program in your own style and process. So I now call Forth my personal language.

Till Later

Well I need to get this out of here, so let me say I will talk more about using Forth in dedicated systems and especially about the newest version of F-PC. Till then keep hacking . . . •

Prototype PC Board Companies:

ARIEL Electronics
1285 Forgewood Ave.
Sunnyvale, CA 94089

HiTech Equipment Company
9400 Activity Road
San Diego, CA 92126
(619)566-1892

Instant Board Circuits Corp.
20 Pamaron Way, Suite A
Novato, Ca 94949

(Continued from page 40)

(programmers, languages) available to support the project.

When I checked out controllers for my last project a major concern was cost. Our project was for 20 units (plus a few extras) and so use of EPROM based controllers was a simple decision. I looked at both embedded EPROMS (part of the MCU) and using MPU's with EPROMs. Intel's EPROM based controllers were two to three times the cost of Motorola's. The price of MPUs and factory burned ROM based MCUs was about the same (both Intel and Motorola). To use the much cheaper factory burned MCU's requires runs of 5,000 or more. Our 20 units didn't approach that number by a long sight.

My cost search indicated that Motorola's 68705 series of MCUs was the cheapest and best choice. The next cost concern was board construction and number of features to be supported. In our case 24 I/O lines were needed and three 30¢ chips were used to multiplex one of the 8 bit I/O buses (unit has 4 I/O buses). The version R3 of the 68705 has a built in A/D converter (also saving cost of a separate chip). As you can see a number of features of the device matched our needs perfectly. Our main controller board ended up with 4 chips, 1 opamp, 8 power transistors, and 2 regulators.

This project was ideally suited for us to design and build our own controller board. Our system was limited and so when a couple of different projects came up with larger needs we looked at other ways of solving the problems. These projects had a high software overhead (program would take up more than 4K) and lots of user interfacing. In one case a Z80 based board was constructed, the other is under consideration but use of PC clone boards is high on the possible list.

In the Z80 case the number of I/O devices was limited and easily handled. The amount of computing however required an interrupt driven system with 32K of ROM and 8K of battery backed-up RAM. Total chip count was 16 and no chip cost more than $2.00. In this case construction cost would be under that of PC clones and less than any off the shelf controller units. The newest design under consideration however needs more.

This proposed design contains multiple communications links, complex user interfaces, several different types of I/O interfaces, and three different software packages. The client is developing a interconnected controller system using master, slaves, and sub-slave units. If separate boards were used for each unit the cost would get out of control. What is needed here is one controller which can be adapted for each possible use.

In the past we would have chosen STD bus products. A single type of CPU board would have been interfaced with I/O cards suited for the needed load demands. Except for the I/O cards all the rest of the hardware would be the same. This sameness allows for easy repair, stocking of parts, and cost reductions. The cost per system however would be $200 to $400 each. The cost of PC clone units are currently running under $80. This lower cost has caused many people, including ourselves to consider these over STD bus. If we have to build our own interface boards for the PC clones and add that cost, as well as the power supply cost, it still is under $200 each system.

If we only consider cost it becomes hard not to consider PC clones. When you add all the software tools available for clones and especially if the project is complex, the decision is usually toward clones. Since we haven't actually done this job yet, I can't say what the long term cost will be. Our 68705 system cost was a little higher than planned (about $65 each) when you figure in all the cost associated with two plotting runs on the PCB fabrication (learning Orcad problems). That is why I would like to end this discussion by asking our readers to drop us letters if they have developed systems using clones over building their own. What we are interested in is actual why, what, and how much the final cost was. We are also interested in whether or not you found the overall project easier or harder. Did you do this under DOS or wrote (bought) your own RTOS (real time operating system).

Moving On

Well this has taken up more space than I planned so my last comments will be brief. I have been looking at other full time work since contracting has slowed down lately (actually stopped). I got interviewed by one company and found the job offer hard to refuse. So as you can guess I am now working for a large company that makes entire systems for handling newspaper publishing. The place made 68000 based terminals and I have been hired to maintain the software.

So far I have found this part of their software activities to be every software company's nightmare. There is no documentation on how all the different versions of software are put together (32 versions). No documents as to what parts fit where (158 files in one version alone, about 4MB of source code). Nothing to help me find out where to start. However I have started plugging away at little pieces and think in about 6 weeks I should have enough information to start programming on the systems. Actually 6 weeks is pretty good when you figure it took them over 10 years to create the problem.

I can say they have learned over the years and are now using automatic library utilities in other programming areas. As I get more experience on their software practices and hardware design I will be passing it on to you. For now I can only comment on how, many years ago they could have gone to a Forth based system. When I worked at a Forth software house, I found out about a Forth network system.

This network system enabled programmers to work on mainframes (just what we do) and then download their code to the final system where it is compiled and run. The big advantage is being able to connect to the remote system as a user on the Forth running there. There are no cross compilers here, just Forth running on several different computer types. These programs would work perfectly for this company because the terminals download their operating code whenever they power up. However, all changes are now done on a separate development system (which is dying) and finished code (32 versions of it) is stored on the mainframe.

What Forth would have given them is the ability to use each work station as a development system. The Forth network also allows use of the kernel for debugging. This is far better than trying to use a debugger with limited commands to see what is actually happening. Debuggers only allow patches in the hex code, where as the Forth patches would be the finished product (you can upload the changes onto the mainframe). Forth's virtual system also would have limited the number of different modules needed. But the product has long passed the stage where improvements are possible, so it now my job to maintain the over 300 files of spaghetti code.

New F-PC 3.5

I got to go to the FORTH DAY, put on by the FORTH INTEREST GROUP in Sunnyvale this year. Instead of a big affair like last year they decided to have a small local fun day. We got to hear Tom Zimmer (and others) talk about Forth. Tom wrote F-PC (with help) and so talked about how it is still growing and some of the newest features. I have since been working on a graphics program for VGA using the FPC structure. I am still wringing out my problems, so will talk about this next time. Charles Moore has a new chip he is working on, and it seemed rather strange but then Forth still seems strange to many people.

On the way down with some friends, we got talking about Forth (what else). We all sort of came to the same point (in different ways of course), that Forth is a per-

(Continued on page 38)

The Computer Corner

By Bill Kibler

Changes and more changes. Lots of things to report on and about. There's WESCON, FORTH, 68000, and more.

WESCON

I went to WESCON in San Francisco, partly to just see the city after the quake and partly to see the latest in electronics. The views of the quake damage were almost invisible as we rode the bus to and from the show. The major problem was the hour long traffic jam getting out of the city that night. It will be a long time before traffic into and out of the city becomes reasonable again.

WESCON is the west coast electronic and component show, alternately held in San Francisco and Los Angeles. I go every other year (when it is in S.F.) and this year's show was little different from two years ago. I thought the number of people visiting was down and not as frantic as last time. A few years back it seemed everyone was hot after some new important product. This year was a how to do it for less approach.

I went with a friend and we both got interested in how to build prototypes in our own shops. He works for a small manufacturing concern and they do small runs of PC boards. They send them out for the major work (PCB layout and fabricating). The cost for doing small quantities can be rather high. Our interests were directed at cheaper means of getting those 10 or 20 boards, as well as those first prototypes. I had heard about some products that would be discussed at a presentation called "$60 dollar PCB prototypes" and had planned on catching it.

I goofed and came on the wrong day, but hunted the vendors down anyway. That actually was a big feat – finding them that is, as the show had what seemed like thousands of booths and two locations. We did find them and spent about a hour talking with HiTech Equipment Corp. I liked their ideas (used a 1 gallon etch bath) but found them not much better than many others. The major problem is plating through the holes. Let me elaborate on how these systems work.

There are a number of vendors all doing about the same (a list is supplied at end of article). You enter your design into a computer using one of the PCB programs (each works with a number of vendors output types – usually HPGL or GERBER plot files) and the system lays down the board. The one type we were interested in used pens of resist and plotted directly on the board. They used a very small etching bath with no plating. Their machine also did the drilling, then you had to solder wires for the through holes. This soldering each through hole problem is where we dropped out of the idea.

In reading the papers presented, one company has an idea they are working on that may solve the problem. It uses boards that have the component holes already drilled and plated. They didn't go into many details but I feel it is similar to using prototype boards that have plated holes on .100 inch spacing. They (Ariel Electronics) use a conductive polymer that is extruded onto the board to form the traces between these plated holes. Baking the board sets the trace permanently. Sounds interesting but I need to see their board layout first.

As a small shop we want the real thing without all the work. That means through holes and component holes that are plated like the real ones. So far the only thing close to that (other than Ariel's design) is using rivets. If your design had hundreds of holes, putting rivets in would be a big chore. I am already looking forward to two years from now when this problem will be solved.

We hunted down Orcad's booth and gave them a bad time over not being able to edit part numbers on the PCB. I guess I wasn't the first one with that complaint as the designer had a "oh not another one" look. He stated they did that on purpose. They want to force you to make corrections in the schematic and netlist. My position is that is fine except when you have a quick change or simple board to layout. You see I did a special board last time that had only 6 components and was shown as a part of the main power layout. We had originally intended to do it in other ways but at the last minute I laid out the board in two hours and was done with it.

To do it however required me to use a debugger on the actual file Orcad created for the board. If you look in the file you can find all the ASCII text. It usually takes two sessions as any new components have a # sign where the values go. So the first time you put numbers in place of the # signs. Running Orcad again lets you see where the numbers are and from that you create a table of those numbers and the actual component number you want them to have. Use debug again and change the numbers to the desired component values and you are done. In some ways I feel I can do it faster than their own means of editing text on the board.

Embedded Systems

There were plenty of companies providing support for embedded systems. My friend was quite amazed at all the analyzers, debuggers, and emulators. The PROM burners were probably the only other group with as many booths supporting embedded systems. I found one group doing lots of 64180/Z180 work and hopefully talked them into doing some articles for us.

This whole area of embedded or small CPU systems has been very confusing for our readers. It seems there are two areas of concerns. The first is what do the terms mean. The other problem is which direction to go in choosing a design solution.

The terms get confusing because Motorola likes to use MPU and MCU to describe their small controllers. The terms mean microcontroller unit (MCU) and microprocessor unit (MPU). CPU has always been central processing unit and I guess Motorola felt the term too misleading. MCUs have only I/O (input and output) pins and no hand shaking for talking to outside memory. MPU's however have the hand shaking lines to allow use of external memory. MPU's also will talk to other devices that talk to I/O. MCU's can handle the current needs of I/O, MPU's can't.

With that simple explanation we can start to see answers for the other concerns. When I (or you) lay out the design problem there are several concerns we need to consider. Cost is most always on the top of the list. Number of input and output lines (and their current needs) is usually next on my list. Overall size of the unit would be next, followed by a long list of features that need to be supported. Lastly I look at tools

(Continued on page 39)